IBM VisualAge C++ for OS/2

S25H-6962-00

Open Class Library User's Guide

Version 3.0

IBM VisualAge C++ for OS/2

S25H-6962-00

Open Class Library User's Guide

Version 3.0

Note!

Before using this information and the product it supports, be sure to read the general information under "Notices" on page xix.

First Edition (May 1995)

This edition applies to Version 3.0 of IBM VisualAge C++ for OS/2 (Programs 30H1664, 30H1665, and 30H1666) and to all subsequent releases and modifications until otherwise indicated in new editions. Make sure you are using the correct edition for the level of the product. Consult the latest edition of the applicable IBM system bibliography for current information on this product.

Order publications through your IBM representative or the IBM branch office serving your locality. Publications are not stocked at the address given below.

A form for readers' comments is provided at the back of this publication. If the form has been removed, address your comments to:

IBM Canada Ltd. Laboratory
Information Development
2G/345/1150/TOR
1150 Eglinton Avenue East
North York, Ontario, Canada. M3C 1H7

You can also send your comments by facsimile (attention: RCF Coordinator), or you can send your comments electronically to IBM. Please see "Communicating Your Comments to IBM" for a description of the methods. This page immediately precedes the Readers' Comment Form at the back of this publication.

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

Contents

Notices

Any reference to an IBM licensed program in this publication is not intended to state or imply that only IBM's licensed program may be used. Any functionally equivalent product, program, or service that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product, program, or service. Evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, is the user's responsibility.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Licensing, IBM Corporation, 500 Columbus Avenue, Thornwood, NY, 10594 USA.

This publication may contain examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples may include the names of individuals, companies, brands, and products. All such names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

Programming Interface Information

This book is intended to help you develop applications that use the C++ class libraries provided with VisualAge C++. This publication documents General-Use Programming Interface and Associated Guidance Information provided by VisualAge C++.

General-Use programming interfaces allow the customer to write programs that obtain the services of VisualAge C++.

Trademarks and Service Marks

The following terms are trademarks of the International Business Machines Corporation in the United States or other countries or both:

C Set ++
CUA
IBMLink
Operating System/2
OS/2 Warp
SAA
VisualAge
Common User Access
IBM
Open Class
OS/2
Presentation Manager
Systems Application Architecture
WorkFrame

UNIX is a registered trademark in the United States and other countries licensed exclusively through X/Open Company Limited.

Other company, product, and service names, which may be denoted by a double asterisk (**), may be trademarks or service marks of others.

About This Book

This book gives you guidance on how to use IBM Open Class Library, the comprehensive library of C++ classes that are provided with VisualAge C++. IBM Open Class Library consists of the following groups of classes, described individually as “class libraries” in this book:

- The Complex Mathematics Library
- The I/O Stream Library
- The Collection Class Library
- The Data Type and Exception Class Library
- The Data Access Builder Class Library
- The User Interface Class Library

The book is divided into parts, beginning with an overview of IBM Open Class Library, and followed by a part for each of the class libraries listed above.

Who Should Use This Book

This book is intended for skilled C++ programmers who want to develop portable C++ applications using IBM Open Class Library. Programmers using this book need to understand the concept of classes. For the Collection Class Library, programmers must also be familiar with using C++ templates. Use this book if you want to do any of the following in your C++ programs:

- Manipulate complex numbers (numbers with both a real and an imaginary part)
- Perform input and output to console or disk devices using a typesafe, object-oriented programming approach
- Implement commonly used abstract data types, including sets, maps, sequences, trees, stacks, queues, and sorted or keyed collections
- Manipulate strings with greater ease and flexibility than the standard C++ method of using character pointers and the string functions of the C `string.h` library
- Use date and time information, and apply methods to date and time objects
- Use Data Access Builder generated source code in conjunction with the Data Access Builder class library to access a DB2/2 relational database.
- Simplify the development of portable applications containing graphical user interfaces (GUI).
- Simulate Common User Access (CUA) workplace look and feel and take advantage of Presentation Manager features.

How to Use This Book

This book is divided into the following chapters and parts:

- **Chapter 1, "Introduction to IBM Open Class Library" on page 1** describes the origins, structure, and uses of each of the class libraries, so that you can decide which libraries or classes to learn about.
- **Part 1, "Complex Mathematics" on page 7** reviews the uses of complex numbers, and describes the `complex` and `c_exception` classes. The `complex` class is used to manipulate complex numbers, and the `c_exception` class is used to handle exceptions resulting from complex number computations.
- **Part 2, "The I/O Stream Library" on page 23** describes the organization of the I/O Stream Class Library, gives reasons for using its classes rather than the I/O interface provided by `stdio.h`, and shows a number of detailed examples of how to do input and output to a console or file, how to format output, how to handle input errors, and so on.
- **Part 3, "The Collection Class Library" on page 73** describes the Collection Class Library, and helps you design applications that use its classes. It discusses instantiating collections, using the Collection Class Library member functions on elements or element keys, tailoring collections for performance, determining the cause of compilation errors resulting from your use of the Collection Classes, and other subjects. This part also contains a set of tutorial lessons that you can use to learn Collection Class Library concepts and techniques.
- **Part 4, "Data Type and Exception Class Library" on page 189** describes the string, date, time, exception, and other classes that make up the Data Type and Exception Class Library, and shows you how to write programs that use the wide range of string-handling and other features provided by this library.
- **Part 5, "The Database Access Class Library" on page 231** describes how to use the code you generated with the Data Access Builder tool with the Data Access Builder Class Library to access data in DB2/2 tables.
- **Part 6, "The User Interface Class Library" on page 263** describes the User Interface Class Library, which you can use to create applications that have the Common User Access (CUA) look and feel.

How to Find Class or Function Descriptions

For detailed information on a particular class or member function, see the appropriate part of the *Open Class Library Reference*. If you know what library a class or member function is in, you can turn to the *Open Class Library Reference* section that describes that library. At the beginning of each *Open Class Library Reference* section you will find a list of all classes described in the section, with page references to class descriptions. Each class description in turn includes an alphabetical listing of member functions with page references to individual functions. If you do not know what section to look in (or what class), you can look up the class or method name in the index.

Classes are organized alphabetically within each class library in the *Open Class Library Reference*, except where classes with similar functionality are placed together. Functions and data members are listed alphabetically at the start of each class chapter, and their descriptions are grouped according to their purpose. If a class has more than one version of a function, all versions are described in one place. For the Collection Classes, all functions of flat collections are described in “Flat Collection Member Functions” in the *Open Class Library Reference*, because each of these functions is used by many or all of the Collection Classes.

A Note about Examples

The examples in this book explain elements of the C++ class libraries. They are coded in a simple style. They do not try to conserve storage, check for errors, achieve fast run times, or demonstrate all possible uses of a library, class, or member function.

Icons Used in This Book

The icons in this book let you quickly scan pages for key concepts, examples, cross-references, and other information.

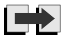
This icon identifies important concepts, programming, and performance tips for using VisualAge C++.

This icon identifies examples that illustrate how to use a particular language feature or other concept presented in the book.

This icon identifies cross-references to related information in this or other books. The icon may appear in the left margin where a number of cross-references are collected, or in miniature form within the text of a paragraph (like this: 📖) where only one or two cross-references are shown.

This icon identifies informa that applies only to Motif** versions of the class library.

This icon identifies informatio that applies only to Presentation Manager versions of the class library.



This icon identifies portability information that you should refer to when you are writing programs that you want to run on multiple platforms.

Related Documentation

See “Bibliography” on page 715 for a list of related books and recommended reading materials.

Introduction to IBM Open Class Library

This book describes IBM Open Class Library, a comprehensive set of C++ class libraries you can use to develop applications:

- The *Complex Mathematics Library* provides you with the facilities to manipulate complex numbers and perform standard mathematical operations on them.
- The *I/O Stream Library* gives you the facilities to deal with many varieties of input and output. You can derive classes from I/O Stream classes to customize the input and output facilities for your own particular needs.
- The *Collection Class Library* provides a set of commonly used abstract data types that you can use to build collections. Collections can have properties such as sorted or unsorted, ordered or unordered, unique-element or multiple-element.
- The *Data Type and Exception Classes* let you manipulate string, date, and time information, and let you handle and trace exceptions.
- The *Data Access Builder Class Library* provides a set of classes and methods that let you connect and disconnect from your DB2/2 database and to perform transactions in the database.
- The *User Interface Class Library* lets you develop portable applications containing graphical user interfaces (GUI) and simulate the Common User Access (CUA) workplace look and feel. You can use these classes to take advantage of Presentation Manager features.

History of IBM Open Class Library

The UNIX** System Laboratories C++ Language System Release 3.0 included Complex, I/O Stream, and Task Libraries. (Earlier releases of this product are known as the AT&T** C++ Language System.) In the Unix System Laboratories product, the class library that corresponds to the I/O Stream Library is called the *Iostream Library*. Prior to Release 2.0 of the AT&T C++ Language System, a class library called the *Stream Library* provided input and output facilities. The I/O Stream Library includes obsolete functions, described in this book, to provide compatibility with the Stream Library.

The Collection Class Library was developed by IBM, as a set of classes for the original C Set ++ for OS/2* product. The classes of the Collection Class Library are exploited by the User Interface Class Library.

Class Library Hierarchies

The Data Type and Exception Classes were developed by IBM, originally as part of the User Interface Class Library on C Set ++ for OS/2.

The User Interface Class Library was developed by IBM, originally for the C Set ++ for OS/2 product. This class library has been significantly enhanced and expanded since its release on C Set ++ for OS/2 Version 2.1.

Hierarchies of the Class Libraries

The following figures show the class hierarchy of the class libraries that make up IBM Open Class. Some of these figures are repeated in the parts that describe specific libraries. For a more detailed description of the Collection Class Library hierarchy figure, see “Abstract Classes” on page 91.

No figure is shown for the Complex Mathematics Library, because the only two classes involved, `complex` and `c_exception`, are not related by inheritance.

Because of the complexity of the User Interface Class Library, no hierarchy diagram is shown for them. For information on the hierarchy of these classes, see the *Open Class Library Reference Volumes 2 and 3*.

The following Data Type and Exception classes are not shown because they do not derive from any class and do not have any subclasses:

- `IExceptionLocation`
- `IMessageText`
- `IStringEnum`
- `IException::TraceFn`
- `IBase::Version`

Class Library Hierarchies

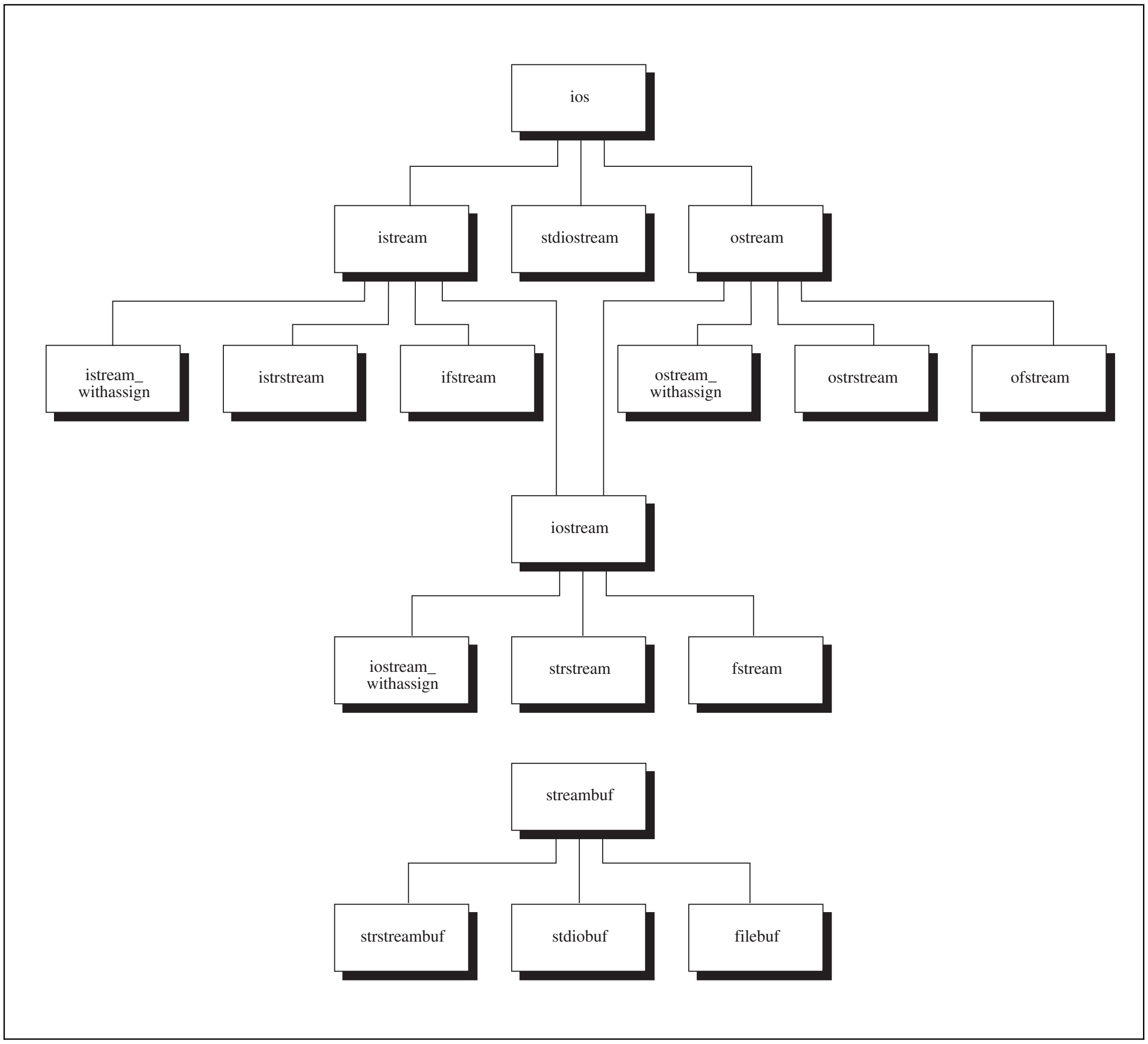


Figure 1. I/O Stream Library Hierarchy

Class Library Hierarchies

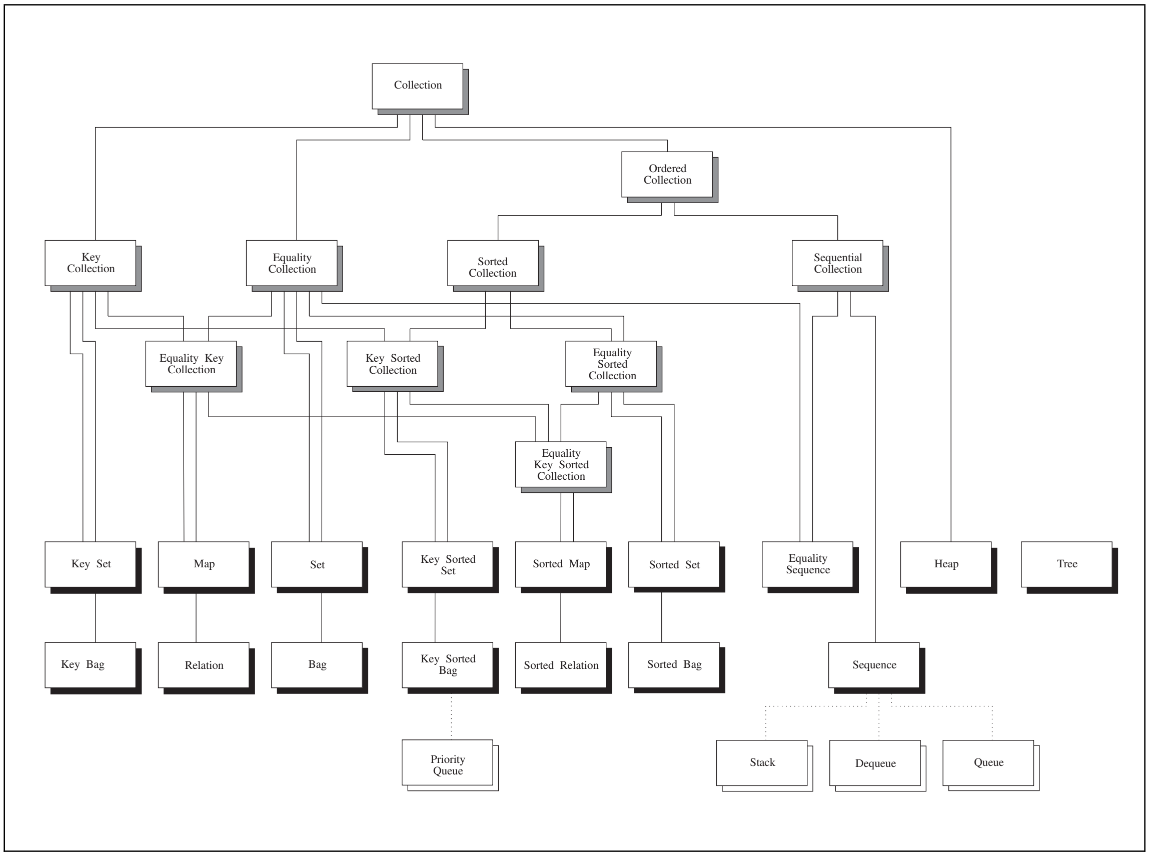


Figure 2. Collection Class Library Hierarchy. Abstract classes have a grey background. Concrete classes have a black background. Restricted access classes have a white background. Dotted lines show a "based-on" relationship, not an actual derivation.

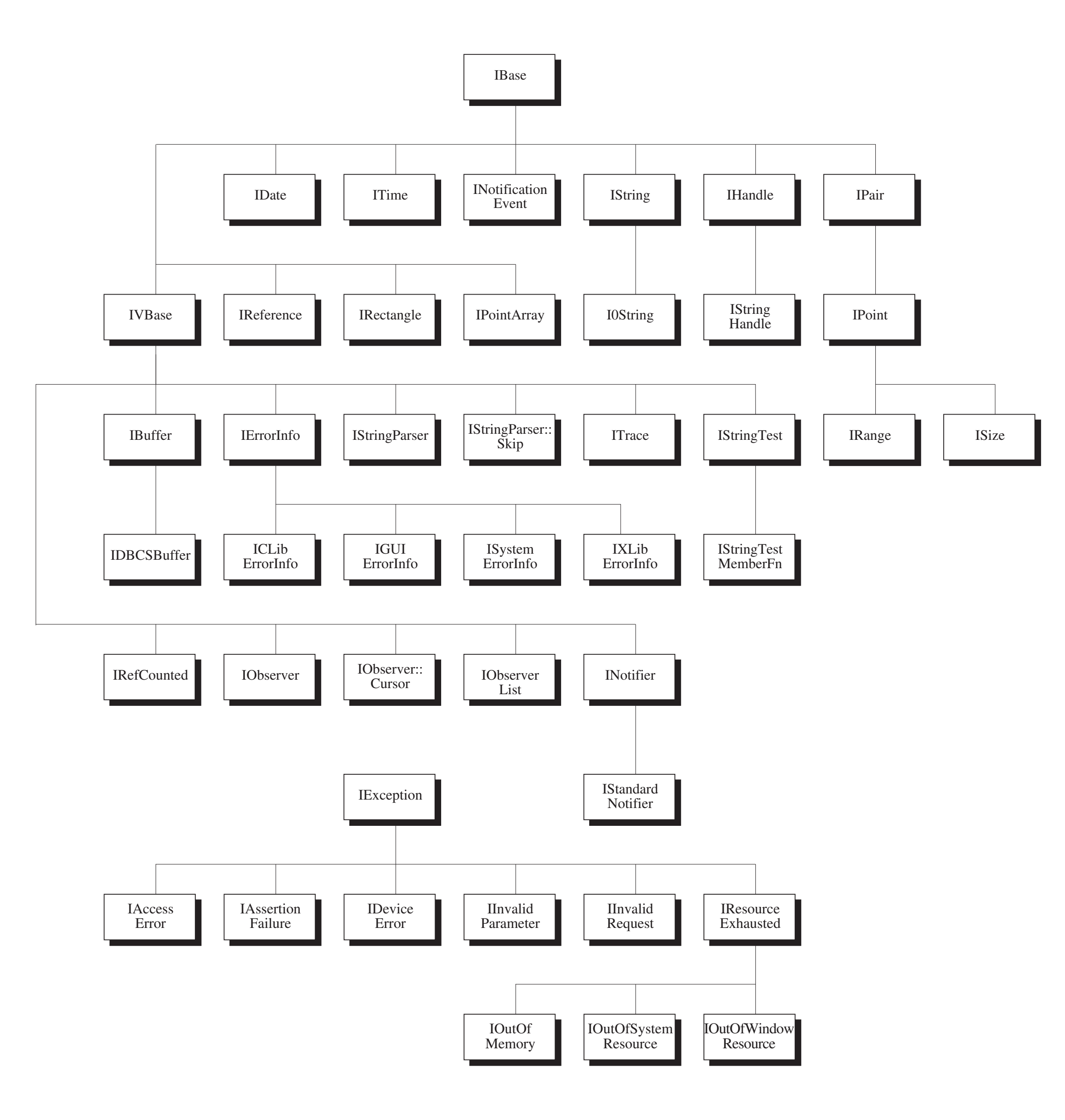


Figure 3. Data Type and Exception Class Hierarchy. Some class names have been split into two lines to fit in their boxes.

Including IBM Open Class Library

A class library is a collection of header and library files. The header files provide the interface to the class libraries.

Including IBM Open Class Library

To use the classes, functions, and operators available in IBM Open Class, you must include the parts of the library's interface that you need in your C++ source program. To include an interface, use the directive `#include <filename>`, where *filename* is the name of the header file. Place this statement at the beginning of the program that requires any of the classes, functions, or operators defined in the header file. Then, in the body of your program, you can use a class, function, or operator defined in the header file, as well as derive new classes and overload the functions and operators.

Part 1. Complex Mathematics

This part provides a review of complex arithmetic, and describes the `complex` and `c_exception` classes.

2 Using the Complex Mathematics Classes

This chapter reviews the concept of complex numbers, and then describes `complex`, the class you use to manipulate complex numbers, and `c_exception`, the class you use to errors created by the functions and operations in the `complex` class. Linking issues, and conflicts between `complex` functions and similarly named functions in the Standard C Runtime Library, are also identified.

Note: The `c_exception` class is not related to the C++ exception handling mechanism that uses the **`try`**, **`catch`**, and **`throw`** statements.

Review of Complex Numbers

A complex number is made up of two parts: a real part and an imaginary part. A complex number can be represented by an ordered pair (a,b), where a is the value of the real part of the number and b is the value of the imaginary part. If (a,b) and (c,d) are complex numbers, then the following statements are true:

- $(a,b) + (c,d) = (a+c,b+d)$
- $(a,b) - (c,d) = (a-c,b-d)$
- $(a,b) * (c,d) = (ac-bd,ad+bc)$
- $(a,b) / (c,d) = ((ac+bd) / (c^2+d^2), (bc-ad) / (c^2+d^2))$
- The conjugate of a complex number (a,b) is $(a,-b)$
- The absolute value or *magnitude* of a complex number (a,b) is the positive square root of the value $a^2 + b^2$
- The polar representation of (a,b) is $(r,theta)$, where r is the distance from the origin to the point (a,b) in the complex plane, and $theta$ is the angle from the real axis to the vector (a,b) in the complex plane. The angle $theta$ can be positive or negative. Figure 4 on page 10 illustrates the polar representation $(r,theta)$ of the complex number (a,b).

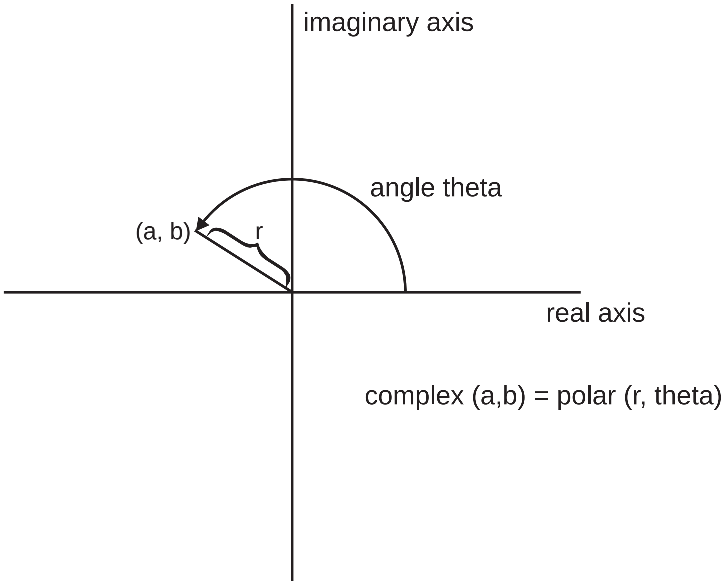


Figure 4. Polar Representation of Complex Number (a,b)

Header Files and Constants for complex and c_exception

You must include the following statement in any file that uses the `complex` or `c_exception` classes:

```
#include <complex.h>
```

This file must be included before any use of the Complex Mathematics Library.

Constants Defined in complex.h

The following table lists the mathematical constants that the Complex Mathematics Library defines (if they have not been previously defined):

Table 1 (Page 1 of 2). Constants Defined in `complex.h`

Constant Name	Description
M_E	The constant e
M_LOG2E	The logarithm of e to the base of 2
M_LOG10E	The logarithm of e to the base of 10
M_LN2	The natural logarithm of 2
M_LN10	The natural logarithm of 10
M_PI	π

Table 1 (Page 2 of 2). Constants Defined in `complex.h`

Constant Name	Description
M_PI_2	π / 2
M_PI_4	π / 4
M_1_PI	1 / π
M_2_PI	2 / π
M_2_SQRTPI	2 divided by the square root of π
M_SQRT2	The square root of 2
M_SQRT1_2	The square root of 1 / 2

Constructing complex Objects

You can use the `complex` constructor to construct initialized or unitialized complex objects or arrays of complex objects. The following example shows different ways of creating and initializing complex objects:

```
complex comp1;                // Initialized to (0, 0)
complex comp2(3.14);          // Initialized to (3.14, 0)
complex comp3(3.14,2.72);     // Initialized to (3.14, 2.72)
complex comparr1[3]={
     1.0,                     // Initialized to (1.0, 0)
     complex(2.0,-2.0),       //                (2.0, -2.0)
     3.0                      //                (3.0, 0)
     };
complex comparr2[3]={
     complex(1.0,1.0),        // Initialized to (1.0, 1.0)
     2.0,                     //                (2.0, 0)
     complex(3.0,-3.0)        //                (3.0, -3.0)
     };
complex comparr3[3]={
     1.0,                     // Initialized to (1.0, 0)
     complex(M_PI_4,M_SQRT2), //                (0.785..., 1.414...)
     M_SQRT1_2                //                (0.707..., 0)
     };
```

Complex Mathematics Input and Output

The `complex` class defines input and output operators for I/O Stream Library input and output. See Part 2, "The I/O Stream Library" on page 23 for more in-depth information on using the I/O Stream Library. Complex numbers are written to the output stream in the format `(`*`real`*`,`*`imag`*`)`. Complex numbers are read from the input stream in one of two formats: `(`*`real`*`,`*`imag`*`)` or *`real`*. The following example shows you how to use the `complex` input and output operators, and provides some sample input and the resulting output.

complex Input and Output

```
//    An example of complex input and output

#include <complex.h>  // required for use of Complex Mathematics Library
#include <iostream.h> // required for use of I/O Stream input and output

void main() {
   complex a[3]={1.0, 2.0, complex(3.0,-3.0)};
   complex b[3];
   complex c[3];
   complex d;

   // read input for all of arrays b and c
   // (you must specify each element individually)
   cout << "Enter three complex values separated by spaces:\n";
   cin >> b[0] >> b[1] >> b[2];

   cout << "Enter three more complex values:\n";
   cin >> c[2] >> c[0] >> c[1];

   // read input for scalar d
   cout << "Enter one more complex value:\n";
   cin >> d;
   // Note that you cannot use the above notation for arrays.
   // For example, cin >> a; is incorrect because a is a complex array.

   // display each array of three complex numbers, then the complex scalar
   cout << "Here are some elements of arrays a, b, and c:\n"
        << a[2] << '\n'
        << b[0] << b[1] << b[2] << '\n'
        << c[1] << '\n'
        << "Here is scalar d: "
        << d << '\n'
   // cout << a produces an address, not a list of array elements:
        << "Here is the address of array a:\n"
        << a
        << endl;     // endl flushes the output stream
}
```

This example produces the output shown below in regular type, given the input shown in bold. Notice that you can insert white space within a complex number, between the brackets, numbers, and comma. However, you cannot insert white space within the real or imaginary part of the number. The address displayed may be different, or in a different format, than the address shown, depending on the operating system, hardware, and other factors.

```
Enter three complex values separated by spaces:
38 (12.2,3.14159) (1712,-33)
Enter three more complex values:
(     17.1234   ,  1234.17) ( 27,    12) (-33    ,0)
Enter one more complex value:
17
Here are some elements of arrays a, b, and c:
( 3, -3)
( 38, 0)( 12.2, 3.14159)( 1712, -33)
( -33, 0)
Here is scalar d: ( 17, 0)
Here is the address of array a:
0x2ff7f9b8
```

Mathematical Operators for complex

The `complex` class defines a set of mathematical operators with the same precedence as the corresponding real operators. With these operators, you can code expressions on complex numbers such as the expressions shown in the example below. In the example, for each complex scalar *x*, the comments showing the results of operations use *xr* to denote the scalar's real part and *xi* to denote the scalar's imaginary part.

```
//   Using the complex mathematical operators

#include <complex.h>
#include <iostream.h>

complex a,b,c,d,e,f,g;

void main() {
cout << "Enter six complex numbers, separated by spaces:\n";
cin >> b >> c >> d >> e >> f >> g;

// assignment, multiplication, addition
a=b*c+d;        // a=( (br*cr)-(bi*ci)+dr , (br*ci)+(bi*cr)+di )

// division
a=b/d;          // a=( (br*dr)+(bi*di) / ((br*br)+(bi*bi),
                //     (bi*dr)-(br*di) / ((br*br)+(bi*bi) )

// subtraction
a=b-f;          // a=( (br-fr), (bi-fi) )

// equality, multiplication assignment
if (a==f) c*=e; // same as c=c*e;

// inequality, addition assignment
if (b!=f) d+=g; // same as d=d+g;

cout << "Here are the seven numbers after calculations:\n"
     << "a=" << a << '\n'
     << "b=" << b << '\n'
     << "c=" << c << '\n'
     << "d=" << d << '\n'
     << "e=" << e << '\n'
     << "f=" << f << '\n'
     << "g=" << g << endl;
}
```

This example produces the output shown below in regular type, given the input shown in bold:

```
Enter six complex numbers, separated by spaces:
(1.14,2.28) (2.24,4.48) (1.17,12.18)
(4.4444444,5.12341) (12,7) 5
Here are the seven numbers after calculations:
a=( -10.86, -4.72)
b=( 1.14, 2.28)
c=( 2.24, 4.48)
d=( 6.17, 12.18)
e=( 4.44444, 5.12341)
f=( 12, 7)
g=( 5, 0)
```

Note that there are no increment or decrement operators for complex numbers.

Mathematical Operators for complex

Equality and Inequality Operators Test for Absolute Equality

The equality and inequality operators test for an exact equality between the real parts of two numbers, and between their complex parts. Because both components are `double` values, two numbers may be “equal” within a certain tolerance, but unequal as far as these operators are concerned. If you want an equality or inequality operator that can test for an absolute difference within a certain tolerance between the two pairs of corresponding components, you should define your own equality functions rather than use the equality and inequality operators of the `complex` class. The functions `is_equal` and `is_not_equal` in the following example provide a reliable comparison between two complex values:

```
// Testing complex values for equality within a certain tolerance

#include <complex.h>
#include <iostream.h>           // for output
#include <iomanip.h>            // for use of setw() manipulator

int is_equal(const complex &a, const complex &b,
             const double tol=0.0001)
{
     return (abs(real(a) - real(b)) < tol &&
             abs(imag(a) - imag(b)) < tol);
}

int is_not_equal(const complex &a, const complex &b,
             const double tol=0.0001)
{
     return !is_equal(a, b, tol);
}

void main()
{
   complex c[4] = { complex(1.0, 2.0),
                    complex(1.0, 2.0),
                    complex(3.0, 4.0),
                    complex(1.0000163,1.999903581) };
   cout << "Comparison of array elements c[0] to c[3]\n"
        << "== means identical,\n!= means unequal,\n"
        << " ~ means equal within tolerance of 0.0001.\n\n"
        << setw(10) << "Element"
        << setw(6)  << 0
        << setw(6)  << 1
        << setw(6)  << 2
        << setw(6)  << 3
        << endl;
   for (int i=0;i<4;i++) {
      cout << setw(10) << i;
      for (int j=0;j<4;j++) {
         if (c[i]==c[j]) cout << setw(6) << "==";
         else if (is_equal(c[i],c[j])) cout << setw(6) << "~";
            else if (is_not_equal(c[i],c[j])) cout << setw(6) << "!=";
               else cout << setw(6) << "???";
          }
      cout << endl;
      }
}
```

This example produces the following output:

```
Comparison of array elements c[0] to c[3]
== means identical,
!= means unequal,
 ~ means equal within tolerance of 0.0001.

   Element     0     1     2     3
         0    ==    ==    !=     ~
         1    ==    ==    !=     ~
         2    !=    !=    ==    !=
         3     ~     ~    !=    ==
```

Assignment Operators Do Not Produce an lvalue

The complex mathematical assignment operators (+=, -=, *=, /=) do not produce a value that can be used in an expression. The following code, for example, produces a compile-time error:

```
   complex x, y, z;     // valid declaration
   x = (y += z);        // invalid assignment causes a
                        // compile-time error
```

Friend Functions for complex

The `complex` class defines a set of mathematical, trigonometric, magnitude, and conversion functions as friend functions of `complex` objects. Because these functions are friend functions rather than member functions, you cannot use the dot or arrow operators. For example:

```
   complex a,b,*c;
   a=exp(b);    // correct - exp() is a friend function of complex
   a=b.exp();   // error - exp() is not a member function of complex
   a=c->exp();  // error - exp() is not a member function of complex
   }
```

Mathematical Functions for complex

The `complex` class defines four mathematical functions as friend functions of `complex` objects. The functions, described in detail in the *Open Class Library Reference*, are:

- exp - Exponent
- log - Logarithm
- pow - Power
- sqrt - Square Root

Friend Functions for complex

The following example shows uses of these mathematical functions:

```
// Using the complex mathematical functions

#include <complex.h>
#include <iostream.h>

void main() {
     complex a, b;
     int i;
     double f;
     //
     // prompt the user for an argument for calls to
     // exp(), log(), and sqrt()
     //
     cout << "Enter a complex value\n";
     cin >> a;
     cout << "The value of exp() for " << a << "  is: " << exp(a)
          << "\nThe natural logarithm of " << a << " is: " << log(a)
          << "\nThe square root of " << a << " is: " << sqrt(a) << "\n\n";
     //
     // prompt the user for arguments for calls to pow()
     //
     cout << "Enter 2 complex values (a and b), an integer (i),"
          << " and a floating point value (f)\n";
     cin >> a >> b >> i >> f;
     cout << "a is " << a << ", b is " << b << ", i is " << i
          << ", f is " << f << '\n'
          << "The value of f**a is: " << pow(f, a) << '\n'
          << "The value of a**i is: " << pow(a, i) << '\n'
          << "The value of a**f is: " << pow(a, f) << '\n'
          << "The value of a**b is: " << pow(a, b) << endl;
     }
```

This example produces the output shown below in regular type, given the input shown in bold:

```
Enter a complex value
(3.7,4.2)
The value of exp() for ( 3.7, 4.2)  is: ( -19.8297, -35.2529)
The natural logarithm of ( 3.7, 4.2) is: ( 1.72229, 0.848605)
The square root of ( 3.7, 4.2) is: ( 2.15608, 0.973992)

Enter 2 complex values (a and b), an integer (i), and a floating point value (f)
(2.6,9.39) (3.16,1.16) -7 33.16237
a is ( 2.6, 9.39), b is ( 3.16, 1.16), i is -7, f is 33.1624
The value of f**a is: ( 972.681, 8935.53)
The value of a**i is: ( -1.13873e-07, -3.77441e-08)
The value of a**f is: ( 4.05451e+32, -4.60496e+32)
The value of a**b is: ( 262.846, 132.782)
```

Trigonometric Functions for complex

The complex class defines four trigonometric functions as friend functions of complex objects. The functions, described in detail in the *Open Class Library Reference*, are:

- cos - Cosine
- cosh - Hyperbolic cosine
- sin - Sine
- sinh - Hyperbolic sine

The following example shows how you can use some of the complex trigonometric functions:

```
// Complex Mathematics Library trigonometric functions

#include <complex.h>
#include <iostream.h>

void main() {
   complex a  = (M_PI, M_PI_2);  // a = (pi,pi/2)
     // display the values of cos(), cosh(), sin(), and sinh()
     // for (pi,pi/2)
   cout << "The value of cos()  for (pi,pi/2) is: " << cos(a)  << '\n'
        << "The value of cosh() for (pi,pi/2) is: " << cosh(a) << '\n'
        << "The value of sin()  for (pi,pi/2) is: " << sin(a)  << '\n'
        << "The value of sinh() for (pi,pi/2) is: " << sinh(a) << endl;
   }
```

This program produces the following output:

```
The value of cos()  for (pi,pi/2) is: ( 6.12323e-17, 0)
The value of cosh() for (pi,pi/2) is: ( 2.50918, 0)
The value of sin()  for (pi,pi/2) is: ( 1, -0)
The value of sinh() for (pi,pi/2) is: ( 2.3013, 0)
```

Magnitude Functions for complex

The magnitude functions for complex are:

- abs - Absolute value
- norm - Square magnitude

See the *Open Class Library Reference* for further details on these functions.

Conversion Functions for complex

The conversion functions in the Complex Mathematics Library allow you to convert between the polar and standard complex representations of a value and to extract the real and imaginary parts of a complex value.

The complex class provides the following conversion functions as friend functions of complex objects:

- arg - Angle in radians
- conj - Conjugation

- polar - Polar to complex
- real - Extract real part
- imag - Extract imaginary part

The following program shows how you can use the complex conversion functions:

```
// Using the complex conversion functions

#include <complex.h>
#include <iostream.h>

void main() {
     complex a;
     // For a value supplied by the user, display the real part,
     // the imaginary part, and the polar representation.
     cout << "Enter a complex value" << endl;
     cin >> a;
     cout << "The real part of this value is " << real(a) << endl;
     cout << "The imaginary part of this value is " << imag(a) << endl;
     cout << "The polar representation of this value is "
          << "(" << abs(a) << "," << arg(a) << ")" << endl;
     }
```

This example produces the output shown below in regular type, given the input shown in bold:

```
Enter a complex value
(175,162)
The real part of this value is 175
The imaginary part of this value is 162
The polar representation of this value is (238.472,0.746842)
```

Using the c_exception Class to Handle Complex Mathematics Errors

Note: The c_exception class is not related to the C++ exception handling mechanism that uses the **try**, **catch**, and **throw** statements.

The c_exception class lets you handle errors that are created by the functions and operations in the complex class. When the Complex Mathematics Library detects an error in a complex operation or function, it invokes complex_error(). This friend function of c_exception has a c_exception object as its argument. When the function is invoked, the c_exception object contains data members that define the function name, arguments, and return value of the function that caused the error, as well as the type of error that has occurred. The data members are:

```
complex arg1;   // First argument of the error-causing function
complex arg2;   // Second argument of the error-causing function
char* name;     // Name of the error-causing function
complex retval; // Value returned by default definition of complex_error
int type;       // The type of error that has occurred.
```

If you do not define your own complex_error function, complex_error sets the complex return value and the errno error number as defined in Table 2 in the *Open Class Library Reference*.

Defining a Customized complex_error Function

You can either use the default version of `complex_error()` or define your own version of the function. In the following example, `complex_error()` is redefined:

```
//Redefinition of the complex_error function

#include <iostream.h>
#include <complex.h>
#include <float.h>

int complex_error(c_exception &c)
{
     cout << "================" << endl;
     cout << "   Exception    " << endl;
     cout << "type = " << c.type << endl;
     cout << "name = " << c.name << endl;
     cout << "arg1 = " << c.arg1 << endl;
     cout << "arg2 = " << c.arg2 << endl;
     cout << "retval = " << c.retval << endl;
     cout << "================" << endl;
     return 0;
}

void main()
{
     complex c1(DBL_MAX,0);
     complex result;
     result = exp(c1);
     cout << "exp" << c1 << "= " << result << endl;
}
```

This example produces the following output:

```
================
   Exception
type = 3
name = exp
arg1 = ( 1.79769e+308, 0)
arg2 = ( 0, 0)
retval = ( infinity, -infinity)
================
exp( 1.79769e+308, 0)= ( infinity, -infinity)
```

If the redefinition of `complex_error()` in the above code is commented out, the default definition of `complex_error()` is used, and the program produces the following output

```
exp( 1.79769e+308, 0) = ( infinity, -infinity)
```

Compiling a Program that Uses a Customized complex_error Function

If you define your own version of `complex_error`, when you compile your program you must use the `/NOE` linker option.

Errors Handled Outside of the Complex Mathematics Library

There are some cases where member functions of the Complex Mathematics Library call functions in the math library. These calls can cause underflow and overflow conditions that are handled by the `matherr()` function that is declared in the `math.h` header file. For example, the overflow conditions that are caused by the following calls are handled by `matherr()`:

- `exp(complex(DBL_MAX, DBL_MAX))`
- `pow(complex(DBL_MAX, DBL_MAX), INT_MAX)`
- `norm(complex(DBL_MAX, DBL_MAX))`

`DBL_MAX` is the maximum valid `double` value, and is defined in `float.h`. `INT_MAX` is the maximum `int` value, and is defined in `limits.h`.

If you do not want the default error-handling defined by `matherr()`, you should define your own version of `matherr()`.

Linking to the Complex Library

You must specify the following library names when compiling or linking programs that use the Complex Library:

- `COMPLEX.LIB` - for single-threaded programs
- `COMPLEXM.LIB` - for multi-threaded programs.

No dynamically linkable version of this library is provided.

An Example of Using the Complex Mathematics Library

The following example shows how you can use the Complex Mathematics Library to calculate the roots of a complex number. For every positive integer *n*, each complex number *z* has exactly *n* distinct *n*th roots. Suppose that in the complex plane the angle between the real axis and point *z* is θ, and the distance between the origin and the point *z* is *r*. Then *z* has the polar form (*r*, θ), and the *n* roots of *z* have the values:

σ
$\sigma \cdot \omega$
$\sigma \cdot \omega^2$
$\sigma \cdot \omega^3$
.
.
.
$\sigma \cdot \omega^{n-1}$

where ω is a complex number with the value:

$$\omega = (\cos(2\pi/n), \sin(2\pi/n))$$

and σ is a complex number with the value:

$$\sigma = r^{1/n} (\cos(\theta/n), \sin(\theta/n))$$

The following code includes two functions, `get_omega()` and `get_sigma()`, to calculate the values of ω and σ. The user is prompted for the complex value *z* and the value of *n*. After the values of ω and σ have been calculated, the *n* roots of *z* are calculated and printed.

```
// Calculating the roots of a complex number

#include <iostream.h>
#include <complex.h>
#include <math.h>

// Function to calculate the value of omega for a given value of n

complex get_omega(double n) {
     complex omega = complex(cos((2.0*M_PI)/n), sin((2.0*M_PI)/n));
     return omega;
     }

//
// function to calculate the value of sigma for a given value of
// n and a given complex value
//
complex get_sigma(complex comp_val, double n) {
     double rn, r, theta;
     complex sigma;
     r = abs(comp_val);
     theta = arg(comp_val);
     rn = pow(r,(1.0/n));
     sigma = rn * complex(cos(theta/n),sin(theta/n));
     return sigma;
}

void main() {
     double n;
     complex input, omega, sigma;
     //
     // prompt the user for a complex number
     //
     cout << "Please enter a complex number: ";
     cin >> input;
     //
     // prompt the user for the value of n
     //
     cout << "What root would you like of this number? ";
     cin >> n;
     //
     // calculate the value of omega
     //
     omega = get_omega(n);
     cout << "Here is omega " << omega << endl;
     //
     // calculate the value of sigma
     //
     sigma = get_sigma(input,n);
     cout << "Here is sigma " << sigma << '\n'
          << "Here are the " << n << " roots of " << input << endl;
```

Complex Mathematics Library Example

```
        for (int i = 0; i < n ; i++) {
             cout << sigma*(pow(omega,i)) << endl;
        }
   }
```

This example produces the output shown below in regular type, given the input shown in bold:

```
Please enter a complex number: (-7, 24)
What root would you like of this number? 2
Here is omega ( -1, 1.22465e-16)
Here is sigma ( 3, 4)
Here are the 2 roots of ( -7, 24)
( 3, 4)
( -3, -4)
```

Part 2. The I/O Stream Library

This part describes the I/O Stream Library, which you can use to perform a wide range of input and output operations in your C++ programs.

Introduction to the I/O Stream Classes

This chapter describes the overall structure of the I/O Stream Classes. These classes provide you with the facilities to deal with many varieties of input and output.

Linking to the I/O Stream Classes

The I/O Stream Libraries are linked in automatically unless you specify the **`/Gn`** option.

The I/O Stream Classes and stdio.h

In both C++ and C, input and output are described in terms of sequences of characters, or *streams*. The I/O Stream Classes provide the same facilities in C++ that `stdio.h` provides in C, but it also has the following advantages over `stdio.h`:

- The input or extraction (>>) operator and the output or insertion (<<) operator are typesafe. They are also easy to use.
- You can define input and output for your own types or classes by overloading the input and output operators. This gives you a uniform way of performing input and output for different types of data.
- The input and output operators are more efficient than `scanf()` and `printf()`, the analogous C functions defined in `stdio.h`. Both `scanf()` and `printf()` take format strings as arguments, and these format strings have to be parsed at run time. This parsing can be time-consuming. The bindings for the I/O Stream output and input operators are performed at compile time, with no need for format strings. This can improve the readability of input and output in your programs, and potentially the performance as well.

Overview of the I/O Stream Classes

The I/O Stream Classes provide the standard input and output capabilities for C++. In C++, input and output are described in terms of *streams*. The processing of these streams is done at two levels. The first level treats the data as sequences of characters; the second level treats it as a series of values of a particular type.

There are two primary base classes for the I/O Stream Classes:

1. The `streambuf` class and the classes derived from it (`strstreambuf`, `stdiobuf`, and `filebuf`) implement the *stream buffers.* Stream buffers act as temporary repositories for characters that are coming from the *ultimate producers* of input or

are being sent to the *ultimate consumers* of output. See "Stream Buffers" on page 32 for more details.

2. The `ios` class maintains formatting and error-state information for these streams. The classes derived from `ios` implement the formatting of these streams. This formatting involves converting sequences of characters from the stream buffer into values of a particular type and converting values of a particular type into their external display format.

The I/O Stream Classes predefine streams for standard input, standard output, and standard error. See "Predefined Streams" on page 29 for more details on the predefined streams. If you want to open your own streams for input or output, you must create an object of an appropriate I/O Streams class. The `iostream` constructor takes as an argument a pointer to a `streambuf` object. This object is associated with the device, file, or array of bytes in memory that is going to be the ultimate producer of input or the ultimate consumer of output.

Combining Input and Output of Different Types

The I/O Stream Classes overload the input (>>) and output (<<) operators for the built-in types. As a result, you can combine input or output of values with different types in a single statement without having to state the type of the values. For example, you can code an output statement such as:

```
<< aFloat << " " << aDouble << "\n" << aString << endl;
```

without needing to provide type or formatting information for each output.

Input and Output for User-Defined Classes

You can overload the input and output operators for the classes that you create yourself. Once you have overloaded the input and output operators for a class, you can perform input and output operations on objects of that class in the same way that you would perform input and output on `char`, `int`, `double`, and the other built-in types.

See "Defining an Input Operator for a Class Type" on page 50 and "Defining an Output Operator for a Class Type" on page 52 for information on how to define class-type input and output operators.

The I/O Stream Class Hierarchy

The I/O Stream Classes have two base classes, streambuf and ios, that correspond to the two levels of processing described in “Overview of the I/O Stream Classes” on page 25:

- The streambuf class implements *stream buffers*. See “Stream Buffers” on page 32 for information on how and why to use stream buffers. streambuf is the base class for the following classes:
  - strstreambuf
  - stdiobuf
  - filebuf
- The ios class maintains formatting and error state information for streams. Streams are implemented as objects of the following classes that are derived from ios:
  - stdiostream
  - istream
  - ostream

The classes that are derived from ios are themselves base classes:

- istream is the input stream class. It implements stream buffer input, or *input* operations. The following classes are derived from istream:
  - istrstream
  - ifstream
  - istream_withassign
  - iostream
- ostream is the output stream class. It implements stream buffer output, or *output* operations. The following classes are derived from ostream:
  - ostrsteam
  - ofstream
  - ostream_withassign
  - iostream
- iostream is the class that combines istream and ostream to implement input and output to stream buffers. The following classes are derived from iostream:
  - strstream
  - iostream_withassign
  - fstream

Note: The I/O Stream Classes also define other classes, including fstreambase and strstreambase. These classes are meant for the internal use of the I/O Stream Classes. Do not use them directly.

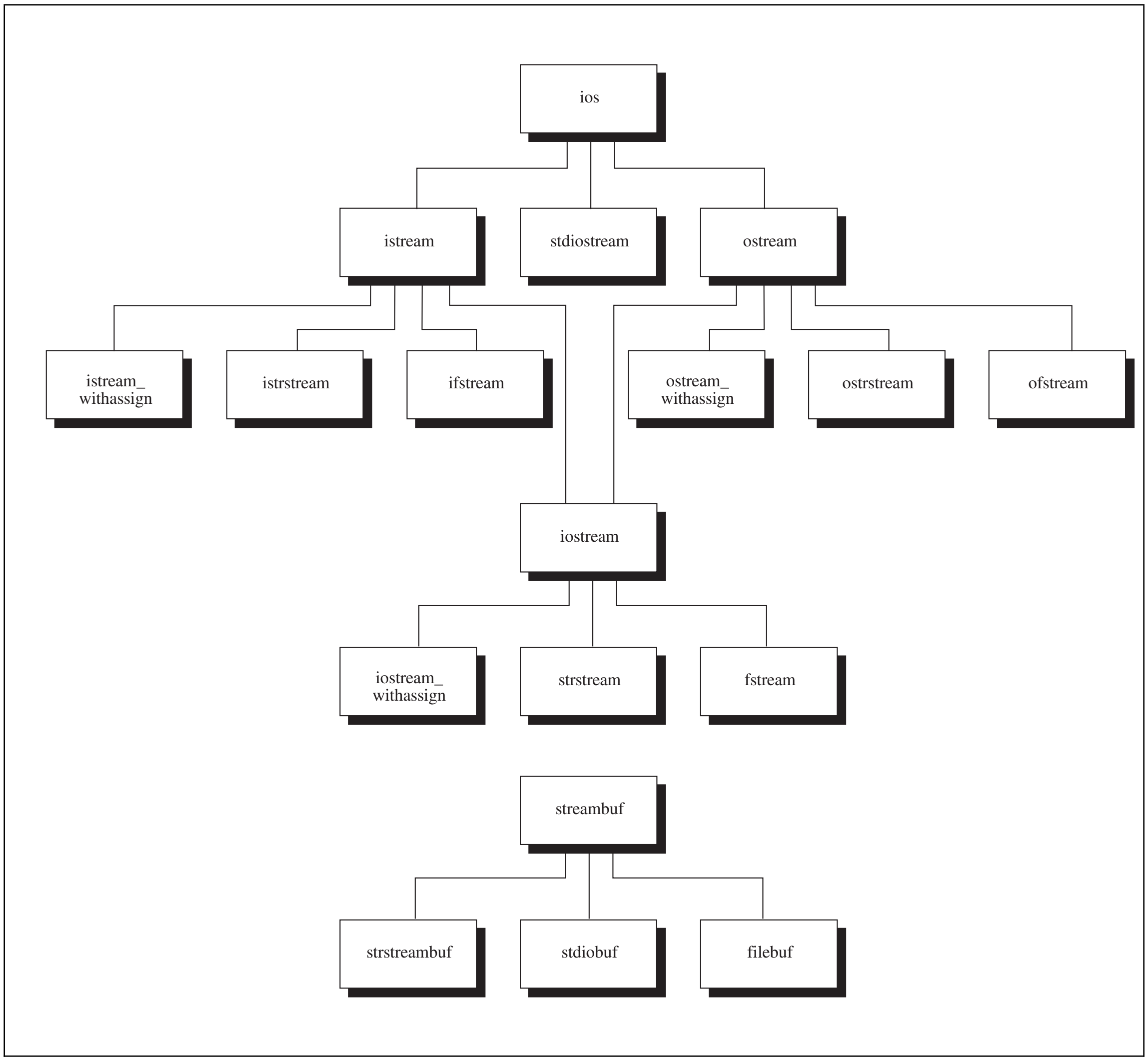


Figure 5. I/O Stream Class Hierarchy

Figure 5 shows the relationship between the two base classes, `ios` and `streambuf`, and their derived classes. In the figure, for any two classes connected by a line, the class at the lower level is derived from the class at the higher level.

The I/O Stream Header Files

To use an I/O Stream class, you must include the appropriate header files for that class. The following lists the I/O Stream header files and the classes that they cover:

- `iostream.h` contains declarations for the basic classes:

  - streambuf
  - ios
  - istream
  - istream_withassign
  - ostream
  - ostream_withassign
  - iostream
  - iostream_withassign

- fstream.h contains declarations for the classes that deal with files:
  - filebuf
  - ifstream
  - ofstream
  - fstream

- stdiostr.h contains declarations for stdiobuf and stdiostream, the classes that specialize streambuf and ios, respectively, to use the FILE structures defined in the C header file stdio.h.

- strstrea.h contains declarations for the classes that deal with character strings. The first "str" in each of these names stands for "string":
  - istrstream
  - ostrsteam
  - strstream
  - strstreambuf

- iomanip.h contains declarations for the parameterized manipulators. Manipulators are values that you can insert into streams or extract from streams to affect or query the behavior of the streams.

- stream.h is used for compatibility with earlier versions of the I/O Stream Classes. It includes iostream.h, fstream.h, stdiostr.h, and iomanip.h, along with some definitions needed for compatibility with the AT&T C++ Language System Release 1.2. Only use this header file with existing code; do not use it with new C++ code.

  Note: If you use the obsolete function form() declared in stream.h, there is a limit to the size of the format specifier. If you call form() with a format specifier string longer than this limit, a runtime message (EDC5091) will be generated and the program will terminate.

Predefined Streams

In addition to giving you the facilities to define your own streams for input and output, the I/O Stream Classes also provide the following predefined streams:

- **cin** is the standard input stream.
- **cout** is the standard output stream.
- **cerr** is the standard error stream. Output to this stream is *unit-buffered.* Characters sent to this stream are flushed after each output operation.
- **clog** is also an error stream, but unlike the output to **cerr**, the output to **clog** is stream-buffered. Characters sent to this stream are flushed only when the stream becomes full or when it is explicitly flushed.

The predefined streams are initialized before the constructors for any static objects are called. You can use the predefined streams in the constructors for static objects.

The predefined streams **cin**, **cerr**, and **clog** are *tied* to **cout**. As a result, if you use **cin**, **cerr**, or **clog**, **cout** is *flushed.* That is, the contents of **cout** are sent to their ultimate consumer. See “tie” in the *Open Class Library Reference* for more details on tying streams together.

Anonymous Streams

An *anonymous stream* is a stream that is created as a temporary object. Because it is a temporary object, an anonymous stream requires a **const** type modifier and is not a modifiable lvalue. Unlike the AT&T C++ Language System Release 2.1, VisualAge C++ does not allow a non-**const** reference argument to be matched with a temporary object. User-defined input and output operators usually accept a non-**const** reference (such as a reference to an istream or ostream object) as an argument. Such an argument cannot be initialized by an anonymous stream, and thus an attempt to use an anonymous stream as an argument to a user-defined input or output operator will usually result in a compile-time error.

In the following example, three methods of writing a character to and reading it from a file are shown:

1. This method uses anonymous streams with the built-in **char** type. This compiles and runs successfully.
2. This method uses anonymous streams with a class that has a **char** as its only data member, and that has input and output operators defined for it. This produces a compilation error if you define anon when you compile. Otherwise, this part of the program is not compiled.
3. This method uses named streams to write a class object to and read it from a file. This compiles and runs successfully.

```
// Using anonymous streams

#include <fstream.h>

class MyClass { public: char a; };
```

```
istream& operator >> (istream& aStream, MyClass mc)
   { return aStream >> mc.a; }

ostream& operator << (ostream& aStream, MyClass mc)
   { return aStream << mc.a; }

void main() {
   char a='a';
   MyClass b,c;
   b.a = 'b';
   c.a = 'c';

// 1 . Use an anonymous stream with a built-in type; this works
   fstream("file1.abc",ios::out) << a << endl;  // write to the file
   fstream("file1.abc",ios::in) >> a;           // read from the file
   cout << a << endl;                           // show what was in the file

#ifdef anon
// 2 . Use an anonymous stream with a class type
// This produces compilation errors if "anon" is defined:

   fstream("file1.abc",ios::out) << b << endl;  // write to the file
   fstream("file1.abc",ios::in) >> b;           // read from the file
   cout << b << endl;                           // show what was in the file
#endif

// 3 . Use a named stream with a class type; this works
   fstream File2("file2.abc",ios::out);         // define and open the file
   File2 << c << endl;                          // write to the file
   File2.close();                               // close the file
   File2.open("file2.abc",ios::in);             // reopen for input
   File2 >> c;                                  // read from the file
   cout << c << endl;                           // show what was in the file
   }
```

If you compile the program with anon defined, compilation fails with messages that resemble the following:

```
Call does not match any argument list for "ostream::operator<<".
Call does not match any argument list for "istream::operator>>".
```

If you compile without anon defined, the letters 'a' and 'c' are written to standard output.

Stream Buffers

One of the most important concepts in the I/O Stream Classes is the stream buffer. The `streambuf` class implements some of the member functions that define stream buffers, but other specialized member functions are left to the classes that are derived from `streambuf`: `strstreambuf`, `stdiobuf`, and `filebuf`.

Note: The AT&T and UNIX System Laboratories C++ Language System documentation use the terms *reserve area* and *buffer* instead of *stream buffer*.

What Does a Stream Buffer Do?

A stream buffer acts as a buffer between the *ultimate producer* (the source of data) or *ultimate consumer* (the target of data) and the member functions of the classes derived from `ios` that format this raw data. The ultimate producer can be a file, a device, or an array of bytes in memory. The ultimate consumer can also be a file, a device, or an array of bytes in memory.

Why Use a Stream Buffer?

In most operating systems, a system call to read data from the ultimate producer or write it to the ultimate consumer is an expensive operation. If your applications can reduce the number of system calls they have to make, they will usually be more efficient. By acting as a buffer between the ultimate producer or ultimate consumer and the formatting functions, a stream buffer can reduce the number of system calls that are made.

Consider, for example, an application that is reading data from the ultimate producer. If there is no buffer, the application has to make a system call for each character that is read. However, if the application uses a stream buffer, system calls will only be made when the buffer is empty. Each system call will read enough characters from the ultimate producer (if they are available) to fill the buffer again.

How Is a Stream Buffer Implemented?

A stream buffer is implemented as an array of bytes. For each stream buffer, pointers are defined that point to elements in this array to define the *get area*, or the space that is available to accept bytes from the ultimate producer, and the *put area*, or the space that is available to store bytes that are on their way to the ultimate consumer.

A stream buffer does not necessarily have separate get and put areas. A stream buffer that is used for input, such as one that is attached to an `istream` object, has a get area. A stream buffer that is used for output, such as one that is attached to an `ostream` object, has a put area. A stream buffer that is used for both input and output, such as one that is attached to an `iostream` object, has both a get area and a put area. In stream buffers implemented by the `filebuf` class that are specialized to use files as an ultimate producer or ultimate consumer, the get and put areas overlap.

The following member functions of the `streambuf` class return pointers to boundaries of areas in a stream buffer:

- `base()` returns a pointer to the beginning of the stream buffer.
- `eback()` returns a pointer to the beginning of the space available for *putback*. Characters that are *putback* are returned to the get area of the stream buffer.
- `gptr()` returns the *get pointer*, a pointer to the beginning of the get area. The space between `gptr()` and `egptr()` has been filled by the ultimate producer. These characters are waiting to be extracted from the stream buffer. The space between `eback()` and `gptr()` is available for *putback*.
- `egptr()` returns a pointer to the end of the get area.
- `pbase()` returns a pointer to the beginning of the space available for the put area.
- `pptr()` returns the *put pointer*, a pointer to the beginning of the put area. The space between `pbase()` and `pptr()` is filled with bytes that are waiting to be sent to the ultimate consumer. The space between `pptr()` and `epptr()` is available to accept characters from the application program that are on their way to the ultimate consumer.
- `epptr()` returns a pointer to the end of the put area.
- `ebuf()` returns a pointer to the end of the stream buffer.

Note: In the actual implementation of stream buffers, the pointers returned by these functions point at `char` values. In the abstract concept of stream buffers, on the other hand, these pointers point to the boundary between `char` values. To establish a correspondence between the abstract concept and the actual implementation, you should think of the pointers as pointing to the boundary just before the character that they actually point at.

Figure 6 on page 34 shows how the pointers returned by these functions delineate the stream buffer.

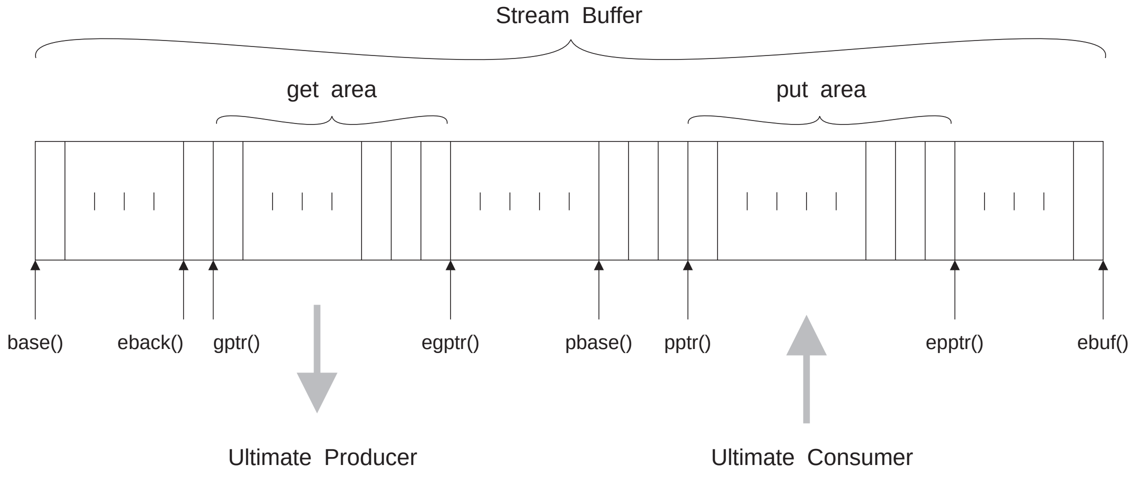


Figure 6. The Structure of Stream Buffers

Format State Flags

The `ios` class defines an enumeration of format state flags that you can use to affect the formatting of data in I/O streams. The following list shows the formatting features and the format flags that control them:

- Whitespace and padding: `ios::skipws`, `ios::left`, `ios::right`, `ios::internal`
- Base conversion: `ios::dec`, `ios::hex`, `ios::oct`, `ios::showbase`
- Integral formatting: `ios::showpos`
- Floating-point formatting: `ios::fixed`, `ios::scientific`, `ios::showpoint`
- Uppercase and lowercase: `ios::uppercase`
- Buffer flushing: `ios::stdio`, `ios::unitbuf`

For examples of how to use these format state flags, see “Changing the Formatting of Stream Output” on page 56. For descriptions of individual format state flags, see “Format State Flags” in the *Open Class Library Reference*.

4 Getting Started with the I/O Stream Library

This chapter identifies common input and output tasks you may want to perform in C++ programs, and shows how you can accomplish these tasks using the I/O Stream Library. The tasks are:

- Receiving input from standard input
- Displaying output on standard output or standard error
- Flushing an output stream with the `endl` and `flush` manipulators
- Parsing multiple inputs
- Opening a file for input and reading from the file
- Opening a file for output and writing to the file.

🕮 If a task you need help with is not listed here, you may find it in Chapter 5, “Advanced I/O Stream Topics” on page 47.

Note: You can compile and run coding examples in this chapter that appear outside of any function, by placing them inside a `main()` function and using `#include <...>` to include necessary header files. Where the header file to include is not indicated, include `iostream.h`.

Receiving Input from Standard Input

When you include the `iostream.h` header file in a program, four streams are automatically defined for I/O use: `cin`, `cout`, `cerr`, and `clog`. The `cin` stream is the standard input stream. Input to `cin` comes from the C standard input stream, `stdin`, unless `cin` has been redirected by the user. The remaining streams can be used for output, and 🕮 their use is described in “Displaying Output on Standard Output or Standard Error” on page 38.

You can receive standard input using the predefined input stream and the input operator (`operator>>`) for the type being read. In the following example, an integer is read from the input stream into a variable:

```
int i;
cin >> i;
```

An input operator must exist for the type being read in. The I/O Stream Library defines input operators for all C++ built-in types. For types you define yourself, you need to provide your own input operators. 🕮 See “Defining an Input Operator for a Class Type” on page 50 for details on how to do this. If you attempt to read input into a variable and no input operator is defined for the type of that variable, the compiler displays an error message with text similar to the following:

```
Call does not match any argument list for "istream::operator>>".
```

Multiple Variables in an Input Statement

You can receive input from a stream into a succession of variables with a single input statement, by repeating the input operator (>>) after each input, and then specifying the next variable to read in. You can combine variables of multiple types in an input statement, without having to specify the types of those variables in the input statement: For example:

```
int i,j,k;
float l,m;
cin >> i >> j >> k >> l >> m;
```

The above syntax provides identical results to the following multiple input statements:

```
int i,j,k;
float l,m;
cin >> i;
cin >> j;
cin >> k;
cin >> l;
cin >> m;
```

If you want to enhance the readability of your source code, break the single input statement up with white space, instead of separating it into multiple input statements:

```
int i,j,k;
float l,m;
cin >> i
    >> j
    >> k
    >> l
    >> m;
```

String Input

If you want to read input into a character array (a string), you should declare the character array using array notation, with a length large enough to hold the largest string being entered. If you declare the character array using pointer notation, you must allocate storage to the pointer, for example by using new or malloc. The following example shows a correct and an incorrect way of placing input in a character array:

```
char goodText[40];
char* badText;
cin >> goodText; // works as long as input is less than 40 chars
cin >> badText;  // may cause a runtime error because no storage
                 // is allocated to *badText
```

In the above example, the input to badText can be made to work by inserting the following code before the input:

```
badText=new char[40];
```

This guideline applies to input to any pointer-to-type: storage must be allocated to the pointer before input occurs.

White Space in String Input

The input operator uses white space to delineate items in the input stream, including strings. If you want an entire line of input to be read in as a single string, you should use the getline() function of istream:

```
// String input using operator << and getline()

#include <iostream.h>

void main() {
   char text1[100], text2[100];

   // prompt and get input for text arrays
   cout << "Enter two words:\n";
   cin >> text1 >> text2;

   // display the text arrays
   cout << "<" << text1 << ">\n"
        << "<" << text2 << ">\n"
        << "Enter two lines of text:\n";

   // ignore the next character if it is a newline
   if (cin.peek()=='\n') cin.ignore(1,'\n');

   // get a line of text into array text1
   cin.getline(text1, sizeof(text1), '\n');

   // get a line of text into array text2
   cin.getline(text2, sizeof(text2), '\n');

   // display the text arrays
   cout << "<" << text1 << ">\n"
        << "<" << text2 << ">" << endl;
   }
```

The first argument of getline() is a pointer to the character array in which to store the input. The second argument specifies the maximum number of bytes of input to read. The third argument is the delimiter, which the library uses to determine when the string input is complete. If you do not specify a delimiter, the default is the new-line character.

Here are two samples of the input and output from this program. Input is shown in bold type, and output is shown in regular type:

```
Enter two words:
Word1 Word2
<Word1>
<Word2>
Enter two lines of text:
First line of text
```

```
Second line of text
<First line of text>
<Second line of text>
```

For the above input, the program works as expected. For the input in the sample below, the first input statement reads two white-space-delimited words from the first line. The check for a new-line character does not find one at the next position (because the next character in the input stream is the space following “happens”), so the first `getline()` call reads in the remainder of the first line of input. The second line of input is read by the second `getline()` call, and the program ends before any further input can be read.

```
Enter two words:
What happens if I enter more words than it asks for?
<What>
<happens>
Enter two lines of text:
I suppose it will skip over the extra ones
< if I enter more words than it asks for?>
<I suppose it will skip over the extra ones>
```

Incorrect Input and the Error State of the Input Stream

When your program requests input through the input operator and the input provided is incorrect or of the wrong type, the error state may be set in the input stream and further input from that input stream may fail. One runtime symptom of such a failure is that your program's prompts for further input display without pausing for the input. See “Correcting Input Stream Errors” on page 54 for details on how to detect and correct input stream errors.

Using Input Streams Other Than cin

You can use the same techniques for input from other input streams as for input from `cin`. The only difference is that, for other input streams, your program must define the stream. For information on how to define an input stream attached to a file, see “Opening a File for Input and Reading from the File” on page 43. Assuming you have defined a stream attached to a file opened for input, and have called that stream `myin`, you can read into that stream from the file by specifying that stream's name instead of `cin`:

```
   // assume the input file is associated with stream myin
   int a,b;
   myin >> a >> b;
```

Displaying Output on Standard Output or Standard Error

The I/O Stream library predefines three output streams as well as the `cin` input stream described in “Receiving Input from Standard Input” on page 35 . The standard output stream is `cout`, and the remaining streams, `cerr` and `clog`, are standard error streams. Output to `cout` goes to the C standard output stream, `stdout`, unless `cout` has been redirected. Output to `cerr` and `clog` goes to the C standard error stream, `stderr`, unless `cerr` or `clog` has been redirected.

`cerr` and `clog` are really two streams that write to the same output device; the difference between them is that `cerr` flushes its contents to the output device after each output, while `clog` must be explicitly flushed.

You can print to one of the predefined output streams by using the predefined stream's name and the output operator (`operator<<`), followed by the value to print:

```
#include <iostream.h>
void main(int argc, char* argv[]) {
   if (argc==1) cout << "Good day!" << endl;
   else cerr << "I don't know what to do with "
            << argv[1] << endl;
}
```

If you name the compiled program `myprog`, the following inputs will produce the following output to standard output or standard error:

Invocation	**Output**
`myprog`	`Good day!` *(to standard output)*
`myprog hello there`	`I don't know what to do with hello` *(to standard error)*

An output operator must exist for any type being output. The I/O Stream Library defines output operators for all C++ built-in types. For types you define yourself, you need to provide your own output operators. See “Defining an Output Operator for a Class Type” on page 52 for details on how to do this. If you attempt to place the contents of a variable into an output stream and no output operator is defined for the type of that variable, the compiler displays an error message with text similar to the following:

```
Call does not match any argument list for "ostream::operator<<".
```

Multiple Variables in an Output Statement

You can place a succession of variables into an output stream with a single output statement, by repeating the output operator (<<) after each output, and then specifying the next variable to output. You can combine variables of multiple types in an output statement, without having to specify the types of those variables in the output statement: For example:

```
int i,j,k;
float l,m;
// ...
cout << i << j << k << l << m;
```

The above syntax provides identical results to the following multiple output statements:

```
int i,j,k;
float l,m;
cout << i;
cout << j;
cout << k;
cout << l;
cout << m;
```

If you want to enhance the readability of your source code, break the single output statement up with white space, instead of separating it into multiple output statements:

```
int i,j,k;
float l,m;
cout << i
     << j
     << k
     << l
     << m;
```

Using Output Streams Other Than cout, cerr, and clog

You can use the same techniques for output to other output streams as for output to the predefined output streams. The only difference is that, for other output streams, your program must define the stream. For information on how to define an output stream attached to a file, see “Opening a File for Output and Writing to the File” on page 46. Assuming you have defined a stream attached to a file opened for output, and have called that stream `myout`, you can write to that file through its stream, by specifying the stream's name instead of `cout`, `cerr` or `clog`:

```
// assume the output file is associated with stream myout
   int a,b;
   myout << a << b;
```

“Opening a File for Output and Writing to the File” on page 46 provides information on all operations required to perform basic file output, including opening, writing to, and closing output files.

Flushing Output Streams with endl and flush

Output streams must be flushed for their contents to be written to the output device. Consider the following:

```
cout << "This first calculation may take a very long time\n";
firstVeryLongCalc();
cout << "This second calculation may take even longer\n";
secondVeryLongCalc();
cout << "All done!"
```

If the functions called in this excerpt do not themselves perform input or output to the standard I/O streams, the first message will be written to the `cout` buffer before `firstVeryLongCalc()` is called. The second message will be written before `secondVeryLongCalc()` is called, but the buffer may not be flushed (written out to the physical output device) until an implicit or explicit flush operation occurs. As a result, the above program displays its messages about expected delays *after* the delays have already occurred. If you want the output to be displayed before each function call, you must flush the output stream.

A stream is flushed implicitly in the following situations:

- The predefined streams `cout` and `clog` are flushed when input is requested from the predefined input stream (`cin`).
- The predefined stream `cerr` is flushed after each output operation.
- An output stream that is unit-buffered is flushed after each output operation. A unit-buffered stream is a stream that has `ios::unitbuf` set. 📖 See “Buffer Flushing” in the *Open Class Library Reference* for further details.
- An output stream is flushed whenever the `flush()` member function is applied to it. This includes cases where the `flush` or `endl` manipulators are written to the output stream. 📖 See “Placing endl or flush in an Output Stream.”
- The program terminates.

The above example can be corrected so that output appears before each calculation begins, as follows:

```
cout << "This first calculation may take a very long time\n";
cout.flush();
firstVeryLongCalc();
cout << "This second calculation may take even longer\n";
cout.flush();
secondVeryLongCalc();
cout << "All done!"
cout.flush();
```

Placing endl or flush in an Output Stream

The `endl` and `flush` manipulators give you a simple way to flush an output stream:

```
cout << "This first calculation may take a very long time" << endl;
firstVeryLongCalc();
cout << "This second calculation may take even longer" << endl;
secondVeryLongCalc();
cout << "All done!" << flush;
```

Placing the `flush` manipulator in an output stream is equivalent to calling `flush()` for that output stream. When you place `endl` in an output stream, it is equivalent to placing a new-line character in the stream, and then calling `flush()`.

Avoid using `endl` where the new-line character is required but buffer flushing is not, because `endl` has a much higher overhead than using the new-line character. For example:

```
cout << "Employee ID:  " << emp.id   << endl
     << "Name:         " << emp.name << endl
     << "Job Category: " << emp.jobc << endl
     << "Hire date:    " << emp.hire << endl;
```

is not as efficient as:

```
cout <<   "Employee ID:  " << emp.id
     << "\nName:         " << emp.name
     << "\nJob Category: " << emp.jobc
     << "\nHire date:    " << emp.hire << endl;
```

You can include the new-line character as the start of the character string that immediately follows the location where the `endl` manipulator would have been placed, or as a separate character enclosed in single quotation marks:

```
cout << "Salary:       " << emp.pay        << '\n'
     << "Next raise:   " << emp.elig_raise << endl;
```

Flushing a stream generally involves a high overhead. If you are concerned about performance, only flush a stream when necessary.

Parsing Multiple Inputs

The I/O Stream Library input streams determine when to stop reading input into a variable based on the type of variable being read and the contents of the stream. The easiest way to understand how input is parsed is to write a simple program such as the following, and run it several times with different inputs.

```
#include <iostream.h>
void main() {
   int a,b,c;
   cin >> a >> b >> c;
   cout << "a: <" << a << ">\n"
        << "b: <" << b << ">\n"
        << "c: <" << c << '>' << endl;
}
```

The following table shows sample inputs and outputs, and explains the outputs. In the “Input” column, `<\n>` represents a new-line character in the input stream.

Input	Output	Remarks
`123<\n>`		No output. `a` has been assigned the value 123, but the program is still waiting on input for `b` and `c`.

`1<\n>`
`2<\n>`
`3<\n>`
	`a: <1>`
	`b: <2>`
	`c: <3>`
		White space (in this case, new-line characters) is used to delimit different input variables.

`1 2  3<\n>`	`a: <1>`
	`b: <2>`
	`c: <3>`
		White space (in this case, spaces) is used to delimit different input variables. There can be any amount of white space between inputs.

`123,456,789<\n>`	`a: <123>`
	`b: <-559038737>`
	`c: <-559038737>`
		Characters are read into `int a` up to the first character that is not acceptable input for an integer (the comma). Characters are read into `int b` where input for `a` left off (the comma). Because a comma is not one of the allowable characters for integer input, `ios::failbit` is set, and all further input fails until `ios::failbit` is cleared. 📖 See "Correcting Input Stream Errors" on page 54 for details on how to clear an input stream.

`1.2 2.3<\n>`
`3.4<\n>`
	`a: <1>`
	`b: <-559038737>`
	`c: <-559038737>`
		As with the previous example, characters are read into `a` until the first character is encountered that is not acceptable input for an integer (in this case, the period). The next input of an `int` causes `ios::failbit` to be set, and so both it and the third input result in errors.

📖 See "White Space in String Input" on page 37 for information on how the input operator interprets white space in the input stream during string input.

Opening a File for Input and Reading from the File

Use the following steps to open a file for input and to read from the file. The steps are described in detail in the subsections that follow the steps.

1. Construct an `fstream` or `ifstream` object to be associated with the file. The file can be opened during construction of the object, or later.
2. Use the name of the `fstream` or `ifstream` object and the input operator or other input functions of the `istream` class, to read the input.
3. Close the file by calling the `close()` member function or by implicitly or explicitly destroying the `fstream` or `ifstream` object.

File Input

Constructing an fstream or ifstream Object for Input

You can open a file for input in one of two ways:

- Construct an fstream or ifstream object for the file, and call open() on the object:

```
#include <fstream.h>
void main() {
   fstream infile1;
   ifstream infile2;
   infile1.open("myfile.dat",ios::in);
   infile2.open("myfile.dat");
   // ...
}
```

- Specify the file during construction, so that open() is called automatically:

```
#include <fstream.h>
void main() {
   fstream infile1("myfile.dat",ios::in);
   ifstream infile2("myfile.dat");
   // ...
}
```

The only difference between opening the file as an fstream or ifstream object is that, if you open the file as an fstream object, you must specify the input mode (ios::in). If you open it as an ifstream object, it is implicitly opened in input mode. The advantage of using ifstream rather than fstream to open an input file is that, if you attempt to apply the output operator to an ifstream object, this error will be caught during compilation. If you attempt to apply the output operator to an fstream object, the error is not caught during compilation, and may pass unnoticed at runtime.

The advantage of using fstream rather than ifstream is that you can use the same object for both input and output. For example:

```
// Using fstream to read from and write to a file

#include <fstream.h>
void main() {
   char q[40];
   fstream myfile("test.x",ios::in); // open the file for input
   myfile >> q;                      // input from myfile into q
   myfile.close();                   // close the file
   myfile.open("test.x",ios::app);   // reopen the file for output
   myfile << q << endl;              // output from q to myfile
   myfile.close();                   // close the file
}
```

This example opens the same file first for input and later for output. It reads in a character string during input, and writes that character string to the end of the same file during output. If the contents of the file `text.x` before the program is run are:

```
barbers often shave
```

the file contains the following after the program is run:

```
barbers often shave
barbers
```

Note that you can use the same `fstream` object to access different files in sequence. In the above example, `myfile.open("test.C",ios::app)` could have read `myfile.open("test.out",ios::app)` and the program would still have compiled and run, although the end result would be that the first string of `test.C` would be appended to `test.out` instead of to `test.C` itself.

Reading Input from a File

The statement `myfile >> a` in the above example reads input into `a` from the `myfile` stream. Input from an `fstream` or `ifstream` object resembles input from the standard input stream `cin`, in all respects except that the input is a file rather than standard input, and you use the `fstream` object name instead of `cin`. The two following programs produce the same output when provided with a given set of input. In the case of `stdin.C`, the input comes from the standard input device. In the case of `filein.C`, the input comes from the file `file.in`:

stdin.C

```
#include <iostream.h>

void main() {
   int ia,ib,ic;
   char ca[40],cb[40],cc[40];
   // cin is predefined
   cin >> ia >> ib >> ic
       >> ca;
   cin.getline(cb,sizeof(cb),'\n');
   cin >> cc;
   // no need to close cin
   cout << ia << ca
        << ib << cb
        << ic << cc << endl;
   }
```

filein.C

```
#include <fstream.h>

void main() {
   int ia,ib,ic;
   char ca[40],cb[40],cc[40];
   fstream myfile("file.in",ios::in);
   myfile >> ia >> ib >> ic
          >> ca;
   myfile.getline(cb,sizeof(cb),'\n');
   myfile >> cc;
   myfile.close();
   cout << ia << ca
        << ib << cb
        << ic << cc << endl;
   }
```

In both examples, the program reads the following, in sequence:

1. Three integers
2. A whitespace-delimited string
3. A string that is delimited either by a new-line character or by a maximum length of 39 characters.
4. A whitespace-delimited string.

Note that, when you define an input operator for a class type, this input operator is available both to the predefined input stream `cin` and to any input streams you define, such as `myfile` in the above example.

For more information on defining your own input operators, see “Defining an Input Operator for a Class Type” on page 50.

For more details on reading input from a stream, see “Receiving Input from Standard Input” on page 35. All techniques for reading input from the standard input stream can be used to read input from a file, providing your code is changed so that the `cin` object is replaced with the name of the `fstream` object associated with the input file.

Opening a File for Output and Writing to the File

The description of using a file as the input stream in “Opening a File for Input and Reading from the File” on page 43 provides the basis for explanations in this section. You may want to read that section first if you have not already done so.

To open a file for output, use the following steps:

1. Declare an `fstream` or `ofstream` object to associate with the file, and open it either when the object is constructed, or later:

   ```
   #include <fstream.h>
   void main() {
      fstream file1("fiup 2out",ios::app);
      ofstream file2("file2.out");
      ofstream file3;
      file3.open("file3.out");
   }
   ```

   You must specify one or more open modes when you open the file, unless you declare the object as an `ofstream` object. Open modes are described in “open” in the *Open Class Library Reference*. The advantage of accessing an output file as an `ofstream` object rather than as an `fstream` object is that the compiler can flag input operations to that object as errors.

2. Use the output operator or `ostream` member functions to perform output to the file.

3. Close the file using the `close()` member function of `fstream`.

When you define an output operator for a class type, this output operator is available both to the predefined output streams and to any output streams you define. For more information on defining your own output operators, see “Defining an Output Operator for a Class Type” on page 52.

5 Advanced I/O Stream Topics

This chapter builds on the information in Chapter 4, "Getting Started with the I/O Stream Library" on page 35, and shows you how to use the I/O Stream Classes to accomplish these more advanced tasks:

- Associating a file with a standard input or output stream
- Using filebuf functions to move through a file
- Defining an input operator for a class type
- Defining an output operator for a class type
- Correcting input stream errors
- Changing the formatting of stream output
- Defining your own format state flags
- Using the strstream classes to accept input from and to send output to character arrays (strings).

If a task you need help with is not listed here, you may find it in Chapter 4, "Getting Started with the I/O Stream Library" on page 35.

Associating a File with a Standard Input or Output Stream

The iostream_withassign class lets you associate a stream object with one of the predefined streams **cin**, **cout**, **cerr**, and **clog**. You can do this, for example, to write programs that accept input from a file if a file is specified, or from standard input if no file is specified.

The following program is a simple filter that reads input from a file into a character array, and writes the array out to a second file. If only one file is specified on the command line, the output is sent to standard output. If no file is specified, the input is taken from standard input. The program uses the iostream_withassign assignment operator to assign an ifstream or ofstream object to one of the predefined streams.

```
//    Generic I/O Stream filter, invoked as follows:
//  filter [infile [outfile] ]

#include <iostream.h>
#include <fstream.h>
void main(int argc, char* argv[])
   {
   ifstream* infile;
   ofstream* outfile;
   char inputline[4096];      // used to read input lines
   int sinl=sizeof(inputline);// used by getline() function
   if (argc>1) {              // if at least an input file was specified
      infile = new ifstream(argv[1]);  // try opening it
      if (infile->good())     // if it opens successfully
         cin = *infile;       // assign input file to cin
      if (argc>2) {             // if an output file was also specified
         outfile = new ofstream(argv[2]);  // try opening it
         if (outfile->good()) // if it opens successfully
            cout = *outfile;  // assign output file to cout
         }
      }

   cin.getline(inputline,
   sizeof(inputline),'\n');             // get first line
   while (cin.good()) {                 // while input is good
   //
   // Insert any line-by-line filtering here
   //
      cout << inputline << endl;        // write line
      cin.getline(inputline,sinl,'\n'); // get next line (sinl specifies
      }                                 // max chars to read)
   if (argc>1) {                        // if input file was used
      infile->close();                  // then close it
      if (argc>2) {                     // if output file was used
         outfile->close();              // then close it
         }
      }
   }
```

You can use this example as a starting point for writing a text filter that scans a file line by line, makes changes to certain lines, and writes all lines to an output file.

Using filebuf Functions to Move Through a File

In a program that receives input from an `fstream` object (a file), you can associate the `fstream` object with a `filebuf` object, and then use the `filebuf` object to move the get or put pointer forward or backward in the file. You can also use `filebuf` member functions to determine the length of the file.

To associate an `fstream` object with a `filebuf` object, you must first construct the `fstream` object and open it. You then use the `rdbuf()` member function of the `fstream` class to obtain the address of the file's `filebuf` object. Using this `filebuf` object, you can move through the file or determine the file's length, with the `seekpos()` and `seekoff()` functions. For example:

```
// Using the filebuf class to move through a file

#include <fstream.h>   // for use of fstream classes
#include <iostream.h>  // not really needed since fstream includes it
#include <stdlib.h>    // for use of exit() function

void main() {
   // declare a streampos object to keep track of the position in filebuf
   streampos Position;

   // declare a streamoff object to set stream offsets
   // (for use by seekoff and seekpos)
   streamoff Offset=0;

   // declare an fstream object and open its file for input
   fstream InputFile("algonq.uin",ios::in);
// check that input was successful, exit if not
if (!InputFile) {
   cerr << "Could not open algonq.uin!  Exiting...\n";
   exit(-1);
   }

// associate the fstream object with a filebuf pointer
filebuf *InputBuffer=InputFile.rdbuf();

// read the first line, and display it
char LineOfFile[128];
InputFile.getline(LineOfFile,sizeof(LineOfFile),'\n');
cout << LineOfFile << endl;

// Now skip forward 100 bytes and display another line
Offset=100;
Position=InputBuffer->seekoff(Offset,ios::cur,ios::in);
InputFile.getline(LineOfFile,sizeof(LineOfFile),'\n');
cout << "At position " << Position << ":\n"
     << LineOfFile << endl;

// Now skip back 50 bytes and display another line
Offset=-50;
Position=InputBuffer->seekoff(Offset,ios::cur,ios::in);
// ios::cur refers to current position in buffer
InputFile.getline(LineOfFile,sizeof(LineOfFile),'\n');
cout << "At position " << Position << ":\n"
     << LineOfFile << endl;

// Now go to position 137 and display to the end of its line
Position=137;
InputBuffer->seekpos(Position,ios::in);
InputFile.getline(LineOfFile,sizeof(LineOfFile),'\n');
cout << "At position " << Position << ":\n"
     << LineOfFile << endl;

// Now close the file and end the program
InputFile.close();
}
```

If the file `algonq.uin` contains the following text:

```
The trip begins on Round Lake.
We proceed through a marshy portage,
and soon find ourselves in a river whose water is the color of ink.

A heron flies off in the distance.
Frogs croak cautiously alongside the canoes.
We can feel the sun's heat glaring at us from grassy shores.
```

the output of the example program is:

```
The trip begins on Round Lake.
At position 131:
ink.
At position 86:
elves in a river whose water is the color of ink.
At position 137:
A heron flies off in the distance.
```

Defining an Input Operator for a Class Type

An input operator is predefined for all built-in C++ types. If you create a class type and want to read input from a file or the standard input device into objects of that class type, you need to define an input operator for that class's type. You define an istream input operator that has the class type as its second argument. For example:

myclass.h

```
#include <iostream.h>

class PhoneNumber {
    public:
       int AreaCode;
       int Exchange;
       int Local;
// Copy Constructor:
       PhoneNumber(int ac, int ex, int lc) :
          AreaCode(ac), Exchange(ex), Local(lc) {}
//... Other member functions
};

istream& operator>> (istream& aStream, PhoneNumber& aPhoneNum) {
   int tmpAreaCode, tmpExchange, tmpLocal;
   aStream >> tmpAreaCode >> tmpExchange >> tmpLocal;
   aPhoneNum=PhoneNumber(tmpAreaCode, tmpExchange, tmpLocal);
   return aStream;
   }
```

The input operator must have the following characteristics:

- Its return type must be a reference to an istream.
- Its first argument must be a reference to an istream. This argument must be used as the function's return value.
- Its second argument must be a reference to the class type for which the operator is being defined.

You can define the code performing the actual input any way you like. In the above example, input is accomplished for the class type by requesting input from the istream object for all data members of the class type, and then invoking the copy constructor for the class type. This is a typical format for a user-defined input operator.

Using the cin Stream in a Class Input Operator

Be careful not to use the cin stream as the input stream when you define an input operator for a class type, unless this is what you really want to do. In the example above, if the line

```
aStream >> tmpAreaCode >> tmpExchange >> tmpLocal;
```

is rewritten as:

```
cin >> tmpAreaCode >> tmpExchange >> tmpLocal;
```

the input operator functions identically, when you use statements in your main program such as cin >> myNumber. However, if the stream requesting input is not the predefined stream cin, then redefining an input operator to read from cin will produce unexpected results. Consider how the following code's behavior changes depending on whether cin or aStream is used as the stream in the input statement within the input operator defined above:

```
#include <iostream.h>
#include <fstream.h>
#include "myclass.h"

void main() {
   PhoneNumber addressBook[40];
   fstream infile("address.txt",ios::in);
   for (int i=0;i<40;i++)
       infile >> addressBook[i]; // does this read from "address.txt"
                                 // or from standard input?
   //...
   }
```

In the original example, the definition of the input operator causes the program to read input from the provided istream object (in this case, the fstream object infile). The input is therefore read from a file. In the example that uses cin explicitly within the input operator, the input that is supposedly coming from infile according to the input statement infile >> addressBook[i] actually comes from the predefined stream cin.

Displaying Prompts in Input Operator Code

You can display prompts for individual data members of a class type within the input operator definition for that type. For example, you could redefine the PhoneNumber input operator shown above as:

```
istream& operator>> (istream& aStream, PhoneNumber& aPhoneNum) {
   int tmpAreaCode, tmpExchange, tmpLocal;
   cout << "Enter area code: ";
   aStream >> tmpAreaCode;
   cout << "Enter exchange:  ";
   aStream >> tmpExchange;
   cout << "Enter local:     ";
   aStream >> tmpLocal;
   aPhoneNum=PhoneNumber(tmpAreaCode, tmpExchange, tmpLocal);
   return aStream;
   }
```

You may be tempted to do this when you anticipate that the source of all input for objects of a class will be the standard input stream **cin**. Avoid this practice wherever possible, because a program using your class may later attempt to read input into an object of your class from a different stream (for example, an fstream object attached to a file). In such cases, the prompts are still written to **cout** even though input from **cin** is not consumed by the input operation. Such an interface does not prevent programs from using your class, but the unnecessary prompts may puzzle end users.

Defining an Output Operator for a Class Type

An output operator is predefined for all built-in C++ types. If you create a class type and want to write output of that class type to a file or to any of the predefined output streams, you need to define an output operator for that class's type. You define an ostream output operator that has the class type as its second argument. For example:

```
// myclass.h
#include <iostream.h>

class PhoneNumber {
    public:
       int AreaCode;
       int Exchange;
       int Local;
// Copy Constructor:
       PhoneNumber(int ac, int ex, int lc) :
          AreaCode(ac), Exchange(ex), Local(lc) {}
//... Other member functions
};

ostream& operator<< (ostream& aStream, PhoneNumber aPhoneNum) {
   aStream << "(" << aPhoneNum.AreaCode << ") "
           << aPhoneNum.Exchange << "-"
           << aPhoneNum.Local << '\n';
   return aStream;
   }
```

The output operator must have the following characteristics:

- Its return type should be a reference to an ostream.
- Its first argument must be a reference to an ostream. This argument must be used as the function's return value.
- Its second argument must be of the class type for which the operator is being defined.

You can define the code performing the actual output any way you like. In the above example, output is accomplished for the class type by placing in the output stream all data members of the class, along with parentheses around the area code, a space before the exchange, and a hyphen between the exchange and the local.

Class Output Operators and the Format State

You should consider checking the state of applicable format flags for any stream you perform output to in a class output operator. At the very least, if you change the format state in your class output operator, before your operator returns it should reset the format state to what it was on entry to the operator. For example, if you design an output operator to always write floating-point numbers at a given precision, you should save the precision in a temporary on entry to your operator, then change the precision and do your output, and reset the precision before returning.

The ios::x_width setting determines the field width for output. Because ios::x_width is reset after each insertion into an output stream (including insertions within class output operators you define), you may want to check the setting of ios::x_width and duplicate it for each output your operator performs. Consider the following example, in which class Coord_3D defines a three-dimensional co-ordinate system. If the function requesting output sets the stream's width to a given value before the output operator for Coord_3D is invoked, the output operator applies that width to each of the three co-ordinates being output. (Note that it lets the width reset after the third output, so that, from the client code's perspective, ios::x_width is reset by the output operation, as it would be for built-in types such as **float**.)

```
// Setting the output width in a class output operator

#include <iostream.h>
#include <iomanip.h>

class Coord_3D {
   public:
      double X,Y,Z;
      Coord_3D(double x, double y, double z) : X(x), Y(y), Z(z)  {}
      };


ostream& operator << (ostream& aStream, Coord_3D coord) {
   int startingWidth=aStream.width();
   aStream << coord.X
#ifndef NOSETW
          << setw(startingWidth)  // set width again
#endif
          << coord.Y
#ifndef NOSETW
          << setw(startingWidth)  // set width again
#endif
          << coord.Z;
   return aStream;
   }

void main() {
   Coord_3D MyCoord(38.162168,1773.59,17293.12);
   cout << setw(17) << MyCoord << '\n'
        << setw(11) << MyCoord << endl;
   }
```

If you add #define NOSETW to prevent the two lines containing setw() in the output operator definition from being compiled, the program produces the output shown below; notice that only the first data member of class Coord_3D is formatted to the desired width.

```
          38.16221773.5917293.1
    38.16221773.5917293.1
```

If you do not comment out the lines containing setw(), all three data members are formatted to the desired width, as shown below:

```
          38.1622          1773.59          17293.1
    38.1622    1773.59    17293.1
```

See “Changing the Formatting of Stream Output” on page 56 for more information on the format state and how to change it, within output operators and in client code.

Correcting Input Stream Errors

When an input statement is requesting input of one type, and erroneous input or input of another type is provided, the error state of the input stream is set to `ios::badbit` and `ios::failbit`, and further input operations may not work properly. For example, the following code repeatedly displays the text: `Enter an integer value:` if the first input provided is a string whose initial characters do not form an integer value:

```
#include <iostream.h>
void main() {
   int i=-1;
   while (i<=0) {
      cout << "Enter a positive integer: " ;
      cin >> i;
      }
   cout << "The value was " << i << endl;
}
```

This program loops indefinitely, given an input such as `ABC12`, because the erroneous input causes the error state to be set in the stream, but does not clear the error state or advance the get pointer in the stream beyond the erroneous characters. Each time the input operator is called for an `int` (as in the `while` loop above), the same characters are read in.

To clear an input stream and repeat an attempt at input you must add code to do the following:

1. Clear the stream's error state.
2. Remove the erroneous characters from the stream.
3. Attempt the input again.

You can determine whether the stream's error state has been set in one of the following ways:

- By calling `fail()` for the stream (shown in the example below)
- By calling `bad()`, `oef()`, `good()`, or `rdstate()`.
- By using the **`void*`** type conversion operator (for example, `if (cin)`).
- By using `operator!` operator (shown in the comment in the example below)

All of these methods are described in “ios Class” on page 31 in the *Open Class Library Reference*.

You can clear the error state by calling `clear()`, and you can remove the erroneous characters using `ignore()`. The example above could be improved, using these suggestions, as follows:

Correcting Input Stream Errors

```
#include <iostream.h>
void main() {
   int i=-1;
   while (i==-1) {
      cout << "Enter a positive integer: ";
      cin >> i;
      while (cin.fail()) {   // could also be "while (!cin) {"
         cin.clear();
         cin.ignore(1000,'\n');
         cerr << "Please try again: ";
         cin >> i;
        }
      }
   cout << "The value was " << i << endl;
}
```

The ignore() member function with the arguments shown above removes characters from the input stream until the total number of characters removed equals 1000, or until the new-line character is encountered, or until EOF is reached. This example produces the output shown below in regular type, given the input shown in bold:

```
Enter an integer value:
ABC12
Please try again:
12ABC
The value was 12
```

Note that, for the second attempt at input, the error state is set *after* the input of 12, so the call to cin.fail() after the corrected input returns false. If another integer input were requested after the **while** loop ends, the error state would be set and that input would fail.

When you define an input operator of class type, you can build error-checking code into your definition. If you do so, you do not have to check for error-causing input every time you use the input operator for objects of your class type. Consider the class definition for the PhoneNumber data type shown in “myclass.h” on page 50, and the following input operator definition:

```
istream& operator>> (istream& aStream, PhoneNumber& aPhoneNum) {
   int AreaCode, Exchange, Local;
   aStream >> AreaCode;
     while (aStream.fail()) eatNonInts(aStream,AreaCode);
   aStream >> Exchange;
     while (aStream.fail()) eatNonInts(aStream,Exchange);
   aStream >> Local;
     while (aStream.fail()) eatNonInts(aStream,Local);
   aPhoneNum=PhoneNumber(AreaCode, Exchange, Local);
   return aStream;
   }
```

The eatNonInts() function in this example should be defined to ignore all characters in the input stream until the next integer character is encountered, and then to read the next integer value into the variable provided as its second argument. The function could be defined as follows:

```
void eatNonInts(istream& aStream, int& anInt) {
   char someChar;
   aStream.clear();
   while (someChar=aStream.peek(), !isdigit(someChar))
      aStream.get(someChar);
   aStream >> anInt;
   }
```

Now whenever input is requested for a `PhoneNumber` object and the provided input contains nonnumeric data, this data is skipped over. Note that this is only a primitive error-handling mechanism; if the input provided is `416 555 2p45` instead of `416 555 2045`, the characters `p45` will be ignored and the local is set to 2 rather than 2045. A more complete example would check each input for the correct number of digits.

Changing the Formatting of Stream Output

The I/O Stream Classes let you define how output should be formatted on a stream-by-stream basis within your program. Most formatting applies to numeric data: what base integers should be written to the output stream in, how many digits of precision floating-point numbers should have, whether they should appear in scientific or fixed-point format. Other formatting applies to any of the built-in types, and to your own types if you design your class output operators to check the format state of a stream to determine what formatting action to take. (See “Defining an Output Operator for a Class Type” on page 52 for suggestions on checking the format state in user-defined output operators.)

This section describes a number of techniques you can use to change the way data is written to output streams. One common characteristic of most of the methods described (other than the method of changing the output field's width) is that each format state setting applies to its output stream until it is explicitly cleared, or is overridden by a mutually exclusive format state. This differs from the C `printf()` family of output functions, in which each `printf()` statement must define its formatting information individually.

ios Methods and Manipulators

For some of the format flags defined for the `ios` class, you can set or clear them using an `ios` function and a flag name, or by using a manipulator. (Manipulators are described in more detail in Chapter 6, “Manipulators” on page 67) With manipulators you can place the change to a stream's state within a list of outputs for that stream. The following example shows two ways of changing the base of an output stream from decimal to octal. The first, which is more difficult to read, uses the `setf()` function to set the `basefield` field in the format state to octal. The second way uses a manipulator, `oct`, within the output statement, to accomplish the same thing:

```
#include <iostream.h>
void main() {
   int a;
   cout.setf(ios::oct,ios::basefield);
   cout << a << endl;
// assume format state gets changed here, so we must change it back
   cout << oct << a << endl;
   }
```

Note that you do not need to qualify a manipulator, provided you do not create a variable or function of the same name as the manipulator. If a variable `oct` were declared at the start of the above example, `cout << oct ...` would write the variable `oct` to standard output. `cout << ios::oct ...` would change the format state.

Using setf, unsetf, and flags

There are two versions of the `setf()` function of `ios`. One version takes a single **`long`** value *`newset`* as argument; its effect is to set any flags set in *`newset`*, without affecting other flags. This version is useful for setting flags that are not mutually exclusive with other flags (for example, `ios::uppercase`). The other version takes two **`long`** values as arguments. The first argument determines what flags to set, and the second argument determines which groups of flags to clear *before* any flags are set. The second argument lets you clear a group of flags before setting one of that group. The second argument is useful for flags that are mutually exclusive. If you try to change the base field of the `cout` output stream using `cout.setf(ios::oct);`, `setf()` sets `ios::oct` but it does not clear `ios::dec` if it is set, so that integers continue to be written to `cout` in decimal notation. However, if you use `cout.setf(ios::oct,ios::basefield);`, all bits in `basefield` are cleared (`oct`, `dec`, and `hex`) before `oct` is set, so that integers are then written to `cout` in octal notation.

To clear format state flags, you can use the `unsetf()` function, which takes a single argument indicating which flags to clear.

To set the format state to a particular combination of flags (without regard for the pre-existing format state), you can use the `flags(long` *`flagset`*`)` member function of `ios`. The value of *`flagset`* determines the resulting values of all the flags of the format state.

The following example demonstrates the use of `flags()`, `setf()`, and `unsetf()`. The `main()` function changes the flags as follows:

1. The original settings of the format state flags are determined, using `flags()`. These settings are saved in the variable `originalFlags`.
2. `ios::fixed` is set, and all other flags are cleared, using `flags(ios::fixed)`.
3. `ios::adjustfield` is set to `ios::right`, without affecting other fields, using `setf(ios::right)`.

4. `ios::floatfield` is set to `ios::scientific`, and `ios::adjustfield` is set to `ios::left`, without affecting other fields. The call to `setf()` is `setf(ios::scientific | ios::left, ios::floatfield|ios::adjustfield)`.
5. The original format state is restored, by calling `flags()` with an argument of `originalFlags`, which contains the format state determined in step 1.

The function `showFlags()` determines and displays the current flag settings. It obtains the value of the settings using `flags()`, and then excludes `ios::oct` from the result before displaying the result in octal. This exclusion is done to display the result in octal without causing the octal setting for `ios::basefield` to show up in the program's output.

```
//Using flags(), flags(long), setf(long), and setf(long,long)

#include <iostream.h>

void showFlags() {
// save altered flag settings, but clear ios::oct from them
      long flagSettings = cout.flags() & (~ios::oct) ;
// display those flag settings in octal
      cout << oct << flagSettings << endl;
}

void main () {
// get and display current flag settings using flags()
      cout << "flags():                              ";
      long originalFlags = cout.flags();
      showFlags();

// change format state using flags(long)
      cout << "flags(ios::fixed):                    ";
      cout.flags(ios::fixed);
      showFlags();

// change adjust field using setf(long)
      cout << "setf(ios::right):                     ";
      cout.setf(ios::right);
      showFlags();

// change floatfield using setf(long, long)
      cout << "setf(ios::scientific | ios::left,\n"
           << "ios::floatfield | ios::adjustfield): ";
      cout.setf(ios::scientific | ios::left,ios::floatfield |ios::adjustfield);
      showFlags();

// reset to original setting
      cout << "flags(originalFlags):                 ";
      cout.flags(originalFlags);
      showFlags();
}
```

This example produces the following output:

```
flags():                             21
flags(ios::fixed):                   10000
setf(ios::right):                    10004
setf(ios::scientific | ios::left,
ios::floatfield | ios::adjustfield): 4002
flags(originalFlags):                21
```

Note:

If you specify conflicting flags, the results are unpredictable. For example, the results will be unpredictable if you set both `ios::left` and `ios::right` in the format state of *`iosobj`*. You should set only one flag in each set of the following three sets:

- `ios::left`, `ios::right`, `ios::internal`
- `ios::dec`, `ios::oct`, `ios::hex`
- `ios::scientific`, `ios::fixed`.

Changing the Notation of Floating-Point Values

You can change the notation and precision of floating-point values to match your program's output requirements. To change the precision with which floating-point values are written to output streams, use `ios::precision()`. By default, an output stream writes `float` and `double` values using six significant digits. The following example changes the precision for the `cout` predefined stream to 17:

```
cout.precision(17);
```

You can also change between scientific and fixed notations for floating-point values. Use the two-parameter version of the `setf()` member function of `ios` to set the appropriate notation. The first argument indicates the flag to be set; the second argument indicates the field of flags the change applies to. For example, to change the notation of an output stream called `File6`, use:

```
File6.setf(ios::scientific,ios::floatfield);
```

This statement clears the settings of the `ios::floatfield` field and then sets it to `ios::scientific`.

The `ios::uppercase` format state variable affects whether the “e” used in scientific-notation floating-point values is in uppercase or lowercase. By default, it is in lowercase. To change the setting to uppercase for an output stream called `TaskQueue`, use:

```
TaskQueue.setf(ios::uppercase);
```

The following example shows the effect on floating-point output of changes made to an output stream using `ios` format state flags and the `precision` member function:

```
// How format state flags and precision() affect output

#include <iostream.h>

void main() {
   double a=3.14159265358979323846;
   double b;
   long originalFlags=cout.flags();
   int originalPrecision=cout.precision();
for (double exp=1.;exp<1.0E+25;exp*=100000000.) {
   cout << "Printing new value for b:\n";
   b=a*exp;     // Initialize b to a larger magnitude of a

// Now print b in a number of ways:
   // In fixed decimal notation
   cout.setf(ios::fixed,ios::floatfield);
   cout << "   " << b << '\n';
   // In scientific notation
   cout.setf(ios::scientific,ios::floatfield);
   cout << "   " <<b << '\n';
   // Change the exponent from lower to uppercase
   cout.setf(ios::uppercase);
   cout << "   " <<b << '\n';
   // With 12 digits of precision, scientific notation
   cout.precision(12);
   cout << "   " <<b << '\n';
   // Same precision, fixed notation
   cout.setf(ios::fixed,ios::floatfield);
   // Now set everything back to defaults
   cout.flags(originalFlags);
   cout.precision(originalPrecision);
   }
}
```

The output from this program is:

```
Printing new value for b:
   3.141593
   3.141593e+00
   3.141593E+00
   3.141592653590E+00
Printing new value for b:
   314159265.358979
   3.141593e+08
   3.141593E+08
   3.141592653590E+08
Printing new value for b:
   31415926535897932.000000
   3.141593e+16
   3.141593E+16
   3.141592653590E+16
Printing new value for b:
   314159265358979280000000.000000
   3.141593e+24
   3.141593E+24
   3.141592653590E+24
```

Changing the Base of Integral Values

For output of integral values, you can choose decimal, hexadecimal, or octal notation. You can either use `setf()` to set the appropriate `ios` flag, or you can place the appropriate parameterized manipulator in the output stream. The following example shows both methods:

```
//Showing the base of integer values

#include <iostream.h>
#include <iomanip.h>
```

```
void main() {
   int a=148;
   cout.setf(ios::showbase);  // show the base of all integral output:
                              //    leading 0x means hexadecimal,
                              //    leading 01 to 07 means octal,
                              //    leading 1 to 9 means decimal
   cout.setf(ios::oct,ios::basefield);
                              // change format state to octal
   cout << a << '\n';
   cout.setf(ios::dec,ios::basefield);
                              // change format state to decimal
   cout << a << '\n';
   cout.setf(ios::hex,ios::basefield);
                              // change format state to hexadecimal
   cout << a << '\n';
   cout << oct << a << '\n';  // Parameterized manipulators clear the
   cout << dec << a << '\n';  // basefield, then set the appropriate
   cout << hex << a << '\n';  // flag within basefield.
   }
```

The `ios::showbase` flag determines whether numbers in octal or hexadecimal notation are written to the output stream with a leading "0" or "0x," respectively. You can set `ios::showbase` where you intend to use the output as input to an I/O Stream input stream later on. If you do not set `ios::showbase` and you try to use the output as input to another stream, octal values and those hexadecimal values that do not contain the digits `a-f` will be interpreted as decimal values; hexadecimal values that do contain any of the digits `a-f` will cause an input stream error.

Setting the Width and Justification of Output Fields

For built-in types, the output operator does not write any leading or trailing spaces around values being written to an output stream, unless you explicitly set the field width of the output stream, using the `width()` member function of `ios` or the `setw()` parameterized manipulator. Both `width()` and `setw()` have only a short-term effect on output. As soon as a value is written to the output stream, the field width is reset, so that once again no leading or trailing spaces are inserted. If you want leading or trailing blanks to appear on successively written values, you can use the `setw()` manipulator within the output statement. For example:

```
#include <iostream.h>
#include <iomanip.h>    // required for use of setw()
void main() {
int i=-5,j=7,k=-9;
cout << setw(5) << i << setw(5) << j << setw(5) << k << endl;
}
```

You can also specify how values should be formatted within their fields. If the current width setting is greater than the number of characters required for the output, you can choose between right justification (the default), left justification, or, for numeric values, internal justification (the sign, if any, is left-justified, while the value is right-justified). For example, the output statement above could be replaced with:

```
cout << setw(5) << i;          // -5
cout.setf(ios::left,ios::adjustfield);
cout << setw(5) << j;          //  7
cout.setf(ios::internal,ios::adjustfield);
cout << setw(5) << k << endl;  // -9
```

The following shows two lines of output, the first from the original example, and the second after the output statement has been modified to use the field justification shown above:

```
-5    7   -9
-57    -   9
```

Defining Your Own Format State Flags

If you have defined your own input or output operator for a class type, you may want to offer some flexibility in how you handle input or output of instances of that class. The I/O Stream Classes let you define stream-specific flags that you can then use with the format state member functions such as `setf()` and `unsetf()`. You can then code checks for these flags in the input and output operators you write for your class types, and determine how to handle input and output according to the settings of those flags.

For example, suppose you develop a program that processes customer names and addresses. In the original program, the postal code for each customer is written to the output file before the country name. However, because of postal regulations, you are instructed to change the record order so that the postal code appears *after* the country name. The following example shows a program that translates from the old file format to the new file format, or from the new file format to the old.

The program checks the input file for an exclamation mark as the first byte. If one is found, the input file is in the new format, and the output file should be in the old format. Otherwise the reverse is true. Once the program knows which file should be in which format, it requests a free flag from each file's stream object. It reads in and writes out each record, and closes the file. The input and output operators for the class check the format state for the defined flag, and order their output accordingly.

```
// Defining your own format flags

#include <fstream.h>
#include <stdlib.h>

long InFileFormat=0;
long OutFileFormat=0;

class CustRecord {
   public:
      int Number;
      char Name[48];
      char Phone[16];
      char Street[128];
      char City[64];
      char Country[64];
      char PostCode[10];
   };

ostream& operator<<(ostream &os, CustRecord &cust) {
   os << cust.Number << '\n'
      << cust.Name   << '\n'
      << cust.Phone  << '\n'
      << cust.Street << '\n'
      << cust.City   << '\n';
   if (os.flags() & OutFileFormat) // New file format
      os << cust.Country << '\n'
         << cust.PostCode << endl;
   else                                  // Old file format
      os << cust.PostCode << '\n'
         << cust.Country << endl;
   return os;
   }

istream& operator>>(istream &is, CustRecord &cust) {
   is >> cust.Number;
   is.ignore(1000,'\n'); // Ignore anything up to and including new line
   is.getline(cust.Name,48);
   is.getline(cust.Phone,16);
   is.getline(cust.Street,128);
   is.getline(cust.City,64);
   if (is.flags() & InFileFormat) { // New file format!
      is.getline(cust.Country,64);
      is.getline(cust.PostCode,10);
      }
   else {
      is.getline(cust.PostCode,10);
      is.getline(cust.Country,64);
      }
   return is;
   }

void main(int argc, char* argv[]) {
   if (argc!=3) {                      // Requires two parameters
      cerr << "Specify an input file and an output file\n";
      exit(1);
      }
   ifstream InFile(argv[1]);
   ofstream OutFile(argv[2],ios::out);

   InFileFormat  =  InFile.bitalloc(); // Allocate flags for
   OutFileFormat = OutFile.bitalloc(); // each fstream

   if (InFileFormat==0 ||              // Exit if no flag could
       OutFileFormat==0) {             // be allocated
      cerr << "Could not allocate a user-defined format flag.\n";
      exit(2);
      }

   if (InFile.peek()=='!') {           // '!' means new format
      InFile.setf(InFileFormat);       // Input file is in new format
      OutFile.unsetf(OutFileFormat);   // Output file is in old format
      InFile.get();                    // Clear that first byte
      }
   else {                              // Otherwise, write '!' to
      OutFile << '!';                  // the output file, set the
      OutFile.setf(OutFileFormat);     // output stream's flag, and
      InFile.unsetf(InFileFormat);     // clear the input stream's
      }                                // flag
```

```
CustRecord record;
while (InFile.peek()!=EOF) {        // Now read the input file
   InFile >> record;                // records and write them
   OutFile << record;               // to the output file,
   }

InFile.close();                     // Close both files
OutFile.close();
}
```

The following shows sample input and output for the program. If you take the output from one run of the program and use it as input in a subsequent run, the output from the later run is the same as the input from the preceding one.

Input File	Output File
3848	!3848
John Smith	John Smith
4163341234	4163341234
35 Baby Point Road	35 Baby Point Road
Toronto	Toronto
M6S 2G2	Canada
Canada	M6S 2G2
1255	1255
Jean Martin	Jean Martin
0418375882	0418375882
48 bis Ave. du Belloy	48 bis Ave. du Belloy
Le Vesinet	Le Vesinet
78110	France
France	78110

Note that, in this example, a simpler implementation could have been to define a global variable that describes the desired form of output. The problem with such an approach is that later on, if the program is enhanced to support input from or output to a number of different streams simultaneously, all output streams would have to be in the same state (as far as the user-defined format variable is concerned), and all input streams would have to be in the same state. By making the user-defined format flag part of the format state of a stream, you allow formatting to be determined on a stream-by-stream basis.

Using the strstream Classes for String Manipulation

You can use the `strstream` classes to perform formatted input and output to arrays of characters in memory. If you create formatted strings using these classes, your code will be less error-prone than if you use the `sprintf()` function to create formatted arrays of characters.

Note: For new applications, you may want to consider using the Data Type class `IString`, rather than `strstream`, to handle strings. The `IString` class provides a much broader range of string-handling capabilities than `strstream`, including the ability to use mathematical operators such as **+** (to concatenate two strings), `=` (to copy one string to another), and `==` (to compare two strings for equality). 🕮 See Chapter 17, “String Classes” on page 197 for further information.

String Manipulation Using strstream

You can use the strstream classes to retrieve formatted data from strings and to write formatted data out to strings. This capability can be useful in situations such as the following:

- Your application needs to send formatted data to an external function that will display, store, or print the formatted data. In such cases, your application, rather than the external function, formats the data.
- Your application generates a sequence of formatted outputs, and requires the ability to change earlier outputs as later outputs are determined and placed in the stream, before all outputs are sent to an output device.
- Your application needs to parse the environment string or another string already in memory, as if that string were formatted input.

You can read input from an strstream, or write output to it, using the same I/O operators as for other streams. You can also write a string to a stream, then read that string as a series of formatted inputs. In the following example, the function add() is called with a string argument containing representations of a series of numeric values. The add() function writes this string to a two-way strstream object, then reads double values from that stream, and sums them, until the stream is empty. add() then writes the result to an ostrstream, and returns OutputStream.str(), which is a pointer to the character string contained in the output stream. This character string is then sent to cout by main().

```
//   Using the strstream classes to parse an argument list

#include <strstream.h>
char* add(char*);

void main() {
   cout << add("1 27 32.12 518") << endl;
   }

char* add(char* addString) {
   double value=0,sum=0;
   strstream TwoWayStream;
   ostrstream OutputStream;
   TwoWayStream << addString << endl;
   for (;;) {
      TwoWayStream >> value;
      if (TwoWayStream) sum+=value;
      else break;
      }
   OutputStream << "The sum is: " << sum << "." << ends;
   return OutputStream.str();
   }
```

This program produces the following output:

```
The sum is: 578.12.
```

String Manipulation Using strstream

6 Manipulators

This chapter introduces manipulators. Manipulators let you change the format state of streams, using the same syntax you use to insert or extract values from those streams.

Introduction to Manipulators

Manipulators provide a convenient way of changing the characteristics of an input or output stream, using the same syntax that is used to insert or extract values. With manipulators, you can embed a function call in an expression that contains a series of insertions or extractions. Manipulators usually provide shortcuts for sequences of `iostream` library operations. See “Simple Manipulators and Parameterized Manipulators” for a description of the two kinds of manipulators.

The `iomanip.h` header file contains a definition for a macro `IOMANIPdeclare()`. `IOMANIPdeclare()` takes a type name as an argument and creates a series of classes you can use to define manipulators for a given kind of stream. Calling the macro `IOMANIPdeclare()` with a type as an argument creates a series of classes that let you define manipulators for your own classes. If you call `IOMANIPdeclare()` with the same argument more than once in a file, you will get a syntax error.

Simple Manipulators and Parameterized Manipulators

There are two kinds of manipulators:

- Simple manipulators do not take any arguments. The following classes have built-in simple manipulators:
  - `ios`
  - `istream`
  - `ostream`
- Parameterized manipulators require one or more arguments. `setfill` (near the bottom of the `iomanip.h` header file) is an example of a parameterized manipulator. You can create your own parameterized manipulators and your own simple manipulators.

The following example shows the uses of both simple and parameterized manipulators. It defines a parameterized manipulator that prints the character <, sets the format state of the output stream to right-justified, and sets the width to the argument with which the manipulator was called. The next output is then right-justified within the specified field width, after the <. The example also defines a simple manipulator that inserts the character > into the output stream, and inserts a new-line and flushes the stream by using the **endl** predefined simple manipulator.

```
// Using simple and parameterized manipulators

#include <iostream.h>
#include <iomanip.h>

ostream& rjust(ostream& os, int n) {     // Parameterized manipulator - set
   os.setf(ios::right,ios::adjustfield); // format flags to right justify,
   return os << '<' << setw(n);          // then print '<', then set width
   }                                     // to manipulator's parameter.

OMANIP(int) rjust(int n) { return OMANIP(int)(rjust,n);}

ostream& endrj (ostream& os) {           // Simple manipulator -- place the
   return os << '>' << endl;             // character '>' in stream, then
   }                                     // a newline character, and flush.

// Notice that, in this example, the simple manipulator uses a
// predefined simple manipulator (endl), while the parameterized
// manipulator uses a predefined parameterized manipulator (setw).

void main() {
   cout << "Employee name:" << rjust(20) << "Sceeles, Darryn" << endrj
        << "Salary:       " << rjust(20) << "$4.25/hour"      << endrj
        << "Next raise:   " << rjust(20) << "9/19/98"         << endrj;
   }
```

This program produces the following output:

```
Employee name:<     Sceeles, Darryn>
Salary:       <          $4.25/hour>
Next raise:   <             9/19/98>
```

Creating Simple Manipulators for Your Own Types

The I/O Stream Library gives you the facilities to create simple manipulators for your own types. Simple manipulators that manipulate `istream` objects are accepted by the following input operators:

```
istream &istream::operator>> (istream&, istream& (*f) (istream&));
istream &istream::operator>> (istream&, ios&(*f) (ios&));
```

Simple manipulators that manipulate `ostream` objects are accepted by the following output operators:

```
ostream &ostream::operator<< (ostream&, ostream&(*f) (ostream&));
ostream &ostream::operator<< (ostream&, ios&(*f) (ios&));
```

The definition of a simple manipulator depends on the type of object that it modifies. The following table shows sample function definitions to modify `istream`, `ostream`, and `ios` objects.

Class of object	Sample function definition
istream	istream &*fi*(istream&){ /*...*/ }
ostream	ostream &*fo*(ostream&){ /*...*/ }
ios	ios &*fios*(ios&){ /*...*/ }

For example, if you want to define a simple manipulator `line` that inserts a line of dashes into an `ostream` object, the definition could look like this:

```
ostream &line(ostream& os) {
   return os << "\n--------------------------------"
             << "--------------------------------\n";
   }
```

Thus defined, the `line` manipulator could be used like this:

```
cout << line << "WARNING! POWER-OUT IS IMMINENT!" << line << flush;
```

This statement produces the following output:

```

----------------------------------------------------------------
WARNING! POWER-OUT IS IMMINENT!
----------------------------------------------------------------
```

Creating Parameterized Manipulators for Your Own Types

The I/O Stream Library gives you the facilities to create parameterized manipulators for your own types. Follow these steps to create a parameterized manipulator that takes an argument of a particular type *tp*:

1. Call the macro `IOMANIPdeclare(`*tp*`)`. Note that *tp* must be a single identifier. For example, if you want *tp* to be a reference to a `long double` value, use `typedef` to make a single identifier to replace the two identifiers that make up the type label `long double`:

   ```
   typedef long double& LONGDBLREF
   ```

2. Determine the class of your manipulator. If you want to define the manipulator as shown in “Example of Defining an APP Parameterized Manipulator” on page 70, choose a class that has `APP` in its name (an `APP` class, also known as an *applicator*). If you want to define the manipulator as shown in “Example of Defining a MANIP Parameterized Manipulator” on page 71, choose a class that has `MANIP` in its name (a `MANIP` class). Once you have determined which type of class to use, the particular class that you choose depends on the type of object that the manipulator is going to manipulate. The following table shows the class of objects to be modified, and the corresponding manipulator classes.

Creating Parameterized Manipulators

Class to be modified	Manipulator class
istream	IMANIP(*tp*) or IAPP(*tp*)
ostream	OMANIP(*tp*) or OAPP(*tp*)
iostream	IOMANIP(*tp*) or IOAPP(*tp*)
The ios part of istream objects or ostream objects	SMANIP(*tp*) or SAPP(*tp*)

3. Define a function *f* that takes an object of the class *tp* as an argument. The definition of this function depends on the class you chose in step 2, and is shown in the following table:

Class chosen	Sample definition
IMANIP(*tp*) or IAPP(*tp*)	istream &f(istream&, tp){/ *... */ }
OMANIP(*tp*) or OAPP(*tp*)	ostream &f(ostream&, tp){/* ... */ }
IOMANIP(*tp*) or IOAPP(*tp*)	iostream &f(iostream&, tp){/* ... */ }
SMANIP(*tp*) or SAPP(*tp*)	ios &f(ios&, tp){/* ... */ }

4. If you chose one of the APP classes in step 2, define the manipulator as shown in “Example of Defining an APP Parameterized Manipulator.” If you chose one of the MANIP classes in step 2, define the manipulator as shown in “Example of Defining a MANIP Parameterized Manipulator” on page 71. These two methods produce equivalent manipulators.

   Note: Parameterized manipulators defined with IOMANIP or IOAPP are not associative. This means that you cannot use such manipulators more than once in a single output statement. See “Examples of Nonassociative Parameterized Manipulators” on page 72 for more details.

Example of Defining an APP Parameterized Manipulator

In the following example, the macro IOMANIPdeclare is called with the user-defined class my_class as an argument. One of the classes that is produced, OAPP(my_class), is used to define the manipulator pre_print.

```
//   Creating and using parameterized manipulators

#include <iomanip.h>

// declare class

class my_class {
public:
     char * s1;
     const char c;
     unsigned short ctr;
     my_class(char *theme, const char suffix,
          unsigned short times):
          s1(theme), c(suffix), ctr(times) {}
};
```

```
   // print a character an indicated number of times
   // followed by a string

   ostream& produce_prefix(ostream& o, my_class mc) {
        for (register i=mc.ctr; i; --i) o << mc.c ;
        o << mc.s1;
        return o;
   }

   IOMANIPdeclare(my_class);

   // define a manipulator for the class my_class

   OAPP(my_class) pre_print=produce_prefix;

   void main() {
      my_class obj("Hello",'-',10);
      cout << pre_print(obj) << endl;
   }
```

This program produces the following output:

```
----------Hello
```

Example of Defining a MANIP Parameterized Manipulator

In the following example, the macro `IOMANIPdeclare` is called with the user-defined class `my_class` as an argument. One of the classes that is produced, `OMANIP(my_class)`, is used to define the manipulator `pre_print()`.

```
#include <iostream.h>
#include <iomanip.h>

class my_class {
public:
     char * s1;
     const char c;
     unsigned short ctr;
     my_class(char *theme, const char suffix,
          unsigned short times):
          s1(theme), c(suffix), ctr(times) {};
};

// print a character an indicated number of times
// followed by a string

ostream& produce_prefix(ostream& o, my_class mc) {
     for (register int i=mc.ctr; i; --i) o << mc.c ;
     o << mc.s1;
     return o;
}

IOMANIPdeclare(my_class);

// define a manipulator for the class my_class

OMANIP(my_class) pre_print(my_class mc) {
     return OMANIP(my_class) (produce_prefix,mc);
}
```

```
void main()
{
        my_class obj("Hello",'-',10);
        cout << pre_print(obj) << "\0" << endl;
}
```

This example produces the same output as the previous example.

Examples of Nonassociative Parameterized Manipulators

The following example demonstrates that parameterized manipulators defined with `IOMANIP` or `IOAPP` are not associative. The parameterized manipulator `mysetw()` is defined with `IOMANIP`. `mysetw()` can be applied once in any statement, but if it is applied more than once, it causes a compile-time error. To avoid such an error, put each application of `mysetw` into a separate statement.

```
// Nonassociative parameterized manipulators

#include <iomanip.h>

iostream&  f(iostream & io, int i) {
     io.width(i);
     return io;
}

IOMANIP (int) mysetw(int i) {
     return IOMANIP(int) (f,i);
}

iostream_withassign ioswa;

void main() {
     ioswa = cout;
     int i1 = 8, i2 = 14;
     //
     // The following statement does not cause a compile-time
     // error.
     //
     ioswa << mysetw(3) << i1 << endl;
     //
     // The following statement causes a compile-time error
     // because the manipulator mysetw is applied twice.
     //
     ioswa << mysetw(3) << i1 <<   mysetw(5) << i2 << endl;
     //
     // The following statements are equivalent to the previous
     // statement, but they do not cause a compile-time error.
     //
     ioswa << mysetw(3) << i1;
     ioswa << mysetw(5) << i2 << endl;
}
```

Part 3. The Collection Class Library

Overview of the Collection Class Library

A C++ collection is an abstract concept, or a C++ class implementing an abstract concept, that allows you to manipulate objects in a group. Collections are used to store and manage elements (or objects) of a user-defined type. Different collections have different internal structures, and different access methods for storage and retrieval of objects.

This chapter describes the types of concrete collections provided by the library, introduces the classes that make up the Collection Class Library, and explains some of the key concepts that are used to describe the Collection Class Library.

Concrete Classes Provided by the Library

This section lists the concrete collections of the Collection Class Library, and provides a verbal description of a potential application of each collection type. These descriptions are also found in the individual class chapters in the Collection Class Library section of the *Open Class Library Reference*. You can use these descriptions to understand the characteristics and behavior of each concrete collection, and of the overall capabilities of the Collection Classes.

Bag An example of using a bag is a program for entering observations on species of plants and animals found in a river. Each time you spot a plant or animal in the river, you enter the name of the species into the collection. If you spot a species twice during an observation period, the species is added twice, because a bag supports multiple elements. You can locate the name of a species that you have observed, and you can determine the number of observations of that species, but you cannot sort the collection by species, because a bag is an unordered collection. If you want to sort the elements of a bag, use a sorted bag instead.

Deque An example of using a deque is a program for managing a lettuce warehouse. Cases of lettuce arriving into the warehouse are registered at one end of the queue (the “fresh” end) by the receiving department. The shipping department reads the other end of the queue (the “old” end) to determine which case of lettuce to ship next. However, if an order comes in for very fresh lettuce, which is sold at a premium, the shipping department reads the “fresh” end of the queue to select the freshest case of lettuce available.

Equality Sequence An example of using an equality sequence is a program that calculates members of the Fibonacci sequence and places them in a collection. Multiple elements of the same value are allowed. For example, the sequence begins with two instances of the value 1. You can search for a given element, for example 8, and find out what element follows it in the sequence. Element equality allows you to do this, using the `locate()` and `setToNext()` functions.

Heap

You can compare using a heap collection to managing the scrap metal entering a scrapyard. Pieces of scrap are placed in the heap in an arbitrary location, and an element can be added multiple times (for example, the rear left fender from a particular kind of car). When a customer requests a certain amount of scrap, elements are removed from the heap in an arbitrary order until the required amount is reached. You cannot search for a specific piece of scrap except by examining each piece of scrap in the heap and manually comparing it to the piece you are looking for.

Key Bag

An example of using a key bag is a program that manages the distribution of combination locks to members of a fitness club. The element key is the number that is printed on the back of each combination lock. Each element also has data members for the club member's name, member number, and so on. When you join the club, you are given one of the available combination locks, and your name, member number, and the number on the combination lock are entered into the collection. Because a given number on a combination lock may appear on several locks, the program allows the same lock number to be added to the collection multiple times. When you return a lock because you are leaving the club, the program finds the elements whose key matches your lock's serial number, and deletes the matching element that has your name associated with it.

Key Set

An example of using a key set is a program that allocates rooms to patrons checking into a hotel. The room number serves as the element's key, and the patron's name is a data member of the element. When you check in at the front desk, the clerk pulls a room key from the board, and enters that key's number and your name into the collection. When you return the key at check-out time, the record for that key is removed from the collection. You cannot add an element to the collection that is already present, because there is only one key for each room. If you attempt to add an element that is already present, the `add()` function returns `False` to indicate that the element was not added.

Key Sorted Bag

An example of using a key sorted bag is a program that maintains a list of families, sorted by the number of family members in each family. The key is the number of family members. You can add an element whose key is already in the collection (because two families can have the same number of members), and you can generate a list of families sorted by size. You cannot locate a family except by its key, because a key sorted bag does not support element equality.

Key Sorted Set

An example of using a key sorted set is a program that keeps track of canceled credit card numbers and the individuals to whom they are issued. Each card number occurs only once, and the collection is sorted by card number. When a merchant enters a customer's card number into a point-of-sale terminal, the collection is checked to see if that card number is listed in the collection of canceled cards. If it is found, the name of the individual is shown, and the merchant is given directions for contacting the card company. If the card number is not found, the transaction can proceed because the card is valid. A list of canceled cards is printed out each month, sorted by card number, and distributed to all merchants who do not have an automatic point-of-sale terminal installed.

Map

An example of using a map is a program that translates integer values between the ranges of 0 and 20 to their written equivalents, from their written forms to their numeric forms. Two maps are created, one with the integer values as keys, one with the written equivalents as keys. You can enter a number, and that number is used as a key to locate the written equivalent. You can enter a written equivalent of a number, and that text is used as a key to locate the value. A given key always matches only one element. You cannot add an element with a key of 1 or “one” if that element is already present in the collection.

Priority Queue

An example of a priority queue is a program used to assign priorities to service calls in a heating repair firm. When a customer calls with a problem, a record with that person's name and the seriousness of the situation is placed in a priority queue. When a service person becomes available, customers are chosen by the program beginning with those whose situation is most severe. In this example, a serious problem such as a nonfunctioning furnace would be indicated by a low value for the priority, and a minor problem such as a noisy radiator would be indicated by a high value for the priority.

Queue

An example of using a queue is a program that processes requests for parts at the cash sales desk of a warehouse. A request for a part is added to the queue when the customer's order is taken, and is removed from the queue when an order picker receives the order form for the part. Using a queue collection in such an application ensures that all orders for parts are processed on a first-come, first-served basis.

Relation

An example of using a relation is a program that maintains a list of all your relatives, with an individual's relationship to you as the key. You can add an aunt, uncle, grandmother, daughter, father-in-law, and so on. You can add an aunt even if an aunt is already in the collection, because you can have several relatives who have the same relationship to you. (For unique relationships such as mother or father, your program would have to check the collection to make sure it did not already contain a family member with that key, before adding the family member.) You can locate a member of the family, but the family members are not in any particular order.

Concrete Classes

Sequence An example of a sequence is a program that maintains a list of the words in a paragraph. The order of the words is obviously important, and you can add or remove words at a given position, but you cannot search for individual words except by iterating through the collection and comparing each word to the word you are searching for. You can add a word that is already present in the sequence, because a given word may be used more than once in a paragraph.

Set An example of a set is a program that creates a packing list for a box of free samples to be sent to a warehouse customer. The program searches a database of in-stock merchandise, and selects ten items at random whose price is below a threshold level. Each item is then added to the set. The set does not allow an item to be added if it is already present in the collection, ensuring that a customer does not get two samples of a single product. The set is not sorted, and elements of the set cannot be located by key.

Sorted Bag An example of using a sorted bag is a program for entering observations on the types of stones found in a riverbed. Each time you find a stone on the riverbed, you enter the stone's mineral type into the collection. You can enter the same mineral type for several stones, because a sorted bag supports multiple elements. You can search for stones of a particular mineral type, and you can determine the number of observations of stones of that type. You can also display the contents of the collection, sorted by mineral type, if you want a complete list of observations made to date.

Sorted Map An example of using a sorted map is a program that matches the names of rivers and lakes to their coordinates on a topographical map. The river or lake name is the key. You cannot add a lake or river to the collection if it is already present in the collection. You can display a list of all lakes and rivers, sorted by their names, and you can locate a given lake or river by its key, to determine its coordinates.

Sorted Relation An example of using a sorted relation is a program used by telephone operators to provide directory assistance. The computerized directory is a sorted relation whose key is the name of the individual or business associated with a telephone number. When a caller requests the number of a given person or company, the operator enters the name of that person or company to access the phone number. The collection can have multiple identical keys, because two individuals or companies might have the same name. The collection is sorted alphabetically, because once a year it is used as the source material for a printed telephone directory.

Sorted Set An example of using a sorted set is a program that tests numbers to see if they are prime. Two complementary sorted sets are used, one for prime numbers, and one for nonprime numbers. When you enter a number, the program first looks in the set of nonprime numbers. If the value is found there, the number is nonprime. If the value is not found there, the program looks in the set of prime numbers. If the value is found there, the number is prime. Otherwise the program determines whether the number is prime or nonprime, and places it in the appropriate sorted set. The program can also display a list of prime or nonprime numbers, beginning at the first prime or nonprime following a given value, because the numbers in a sorted set are sorted from smallest to largest.

Stack An example of using a stack is a program that keeps track of daily tasks that you have begun to work on but that have been interrupted. When you are working on a task and something else comes up that is more urgent, you enter a description of the interrupted task and where you stopped it into your program, and the task is pushed onto the stack. Whenever you complete a task, you ask the program for the most recently saved task that was interrupted. This task is popped off the stack, and you resume your work where you left off. When you attempt to pop an item off the stack and no item is available, you have completed all your tasks.

Benefits of the Collection Class Library

In addition to implementing the common abstract data types efficiently and reliably, the Collection Class Library gives you the following benefits:

- A framework of *properties* to help you decide which abstract data type is appropriate in a given situation
- A choice about how the abstract data type you have chosen is implemented by the Collection Class Library

The Collection Class Library lets you choose the appropriate abstract data type for a given situation by providing *collection classes* that are a complete, systematic, and consistent combination of basic properties. These properties, which are explained in “Flat Collections” on page 80 , help you to select abstract data types that are at the appropriate level of abstraction. In a particular application, for example, you may have the choice between using a bag and a key sorted set. The properties of these two collections will help you decide which one is more appropriate.

Once you have chosen the appropriate abstract data type, the Collection Class Library offers you a choice of implementations for it. Each abstract data type has a common interface with all of its possible implementations. It is easy to replace one implementation with another for performance reasons or if the requirements of your application change.

Types of Classes in the Collection Class Library

The classes that make up the Collection Class Library are divided into three types:

Flat Collections

Flat collections include abstractions such as sequence, set, bag, and map. Unlike trees, flat collections have no hierarchy of elements or recursive structure.

See “Flat Collections” for more information on flat collections and their properties.

Trees

Trees are recursive collections of nodes, where each node holds an element and has a given number of nodes as children.

See “Trees” on page 87 for more details on trees.

Auxiliary Classes

The *auxiliary classes* include classes for cursors, iterators, and simple and managed pointers.

Cursors and iterators give you convenient methods for accessing the elements stored in the collections. See “Cursors” on page 98 for more details on cursor classes. See “Iteration Using Iterators” on page 102 for more details on iterator classes.

The pointer classes provide the means to store in collections a pointer to an object instead of the object itself. The managed pointer class offers this object management together with automatic storage management. See “Using Pointer Classes” on page 115 and “Managed Pointers” on page 119 for more details on pointer classes.

Flat Collections

Four basic properties are used to differentiate between different flat collections:

Ordering

Whether a *next* or *previous* relationship exists between elements.

Access by key

Whether a part of the element (a *key*) is relevant for accessing an element in the collection. When keys are used, they are compared using relational operators.

Equality for elements

Whether equality is defined for the element.

Uniqueness of entries

Whether any given element or key is *unique*, or whether *multiple* occurrences of the same element or key are allowed.

Figure 7 shows the flat collection that results from each combination of properties. For example, “Map” appears in the Unique, Unordered column for the Key, Element Equality row. This means that a map is unordered, each element is unique, keys are defined, and element equality is defined. The figure contains N/A where no flat collection corresponds to the combination of properties. For example, the N/A in the first two rows of the rightmost column indicates that an ordered collection that is sequential instead of sorted and offers access by key is not available. This implies that there are no flat collections that have all of the following properties:

- The collection is ordered.
- The collection is sequential.
- The collection allows an element to appear more than once.
- Keys are defined for elements in the collection.

The rationale for not implementing collections with these combinations of properties is that there is no reason to choose them over another collection that is already available. For example, for an ordered collection that is sequential and offers access by key, the key access would only have advantages if the elements are stored in a position depending on their key. Because they are not, there is no flat collection with key access that maintains a sequential order.

		Unordered		**Ordered**		
				Sorted		**Sequential**
		Unique	**Multiple**	**Unique**	**Multiple**	**Multiple**
Key (Key equality must be defined)	**Element Equality**	Map	Relation	Sorted map	Sorted relation	N/A
	No Element Equality	Key set	Key bag	Key sorted set	Key sorted bag	N/A
No Key	**Element Equality**	Set	Bag	Sorted set	Sorted bag	Equality sequence
	No Element Equality	N/A	Heap	N/A	N/A	Sequence

Figure 7. Combination of Flat Collection Properties

Flat Collections

Ordering of Collection Elements

The elements of a flat collection class can be ordered in three ways:

- *Unordered* collections have elements that are not ordered.
- *Sorted* collections have their elements sorted by an ordering relation defined for the element type. For example, integers can be sorted in ascending order, and strings can be ordered alphabetically. The ordering relation is determined by the instantiations for the collection class. For elements where the ordering relation returns the same position, elements are added in chronological order.
- *Sequential* collections have their ordering determined by an explicit qualifier to the `add()` function, for example, `addAtPosition()`.

A particular element in a sorted collection can be accessed quickly by using the ordering relation to determine its position. Unordered collections can also be implemented to allow fast access to the elements, by using, for example, a hash table or a sorted representation. The Collection Class Library provides a fast `locate()` function that uses this structure for unordered and sorted collections. Even though unordered collections are often implemented by sorting the elements, do not assume that all unordered collections are implemented in this way. If your program requires this assumption to be true, use a sorted collection instead.

For each flat collection, the Collection Class Library provides both unordered and sorted abstractions. For example, the Collection Class Library supports both a set and a sorted set. The ordering property is independent of the other properties of flat collections: you have the choice of making a given flat collection unordered or sorted regardless of the choices that you make for the other properties.

Access by Key

A given flat collection can have a *key* defined for its elements. A key is usually a data member of the element, but it can also be calculated from the data members of the element by some arbitrary function. Keys let you:

- Organize the elements in a collection
- Access a particular element in a collection

For collections that have a key defined, an equality relation must be defined for the key type. Thus, a collection with a key is said to have *key equality*.

Equality for Keys and Elements

A flat collection can have an equality relation defined for its elements. The default equality relation is based on the element as a whole, not just on one or more of its data members (for example, the key). For two elements to be equal, all data members of both elements must be equal. The equality relation is needed for functions such as those that locate or remove a given element. A flat collection that has an equality relation has *element equality*.

Note that, for non-built-in types, you can define your own equality relation to behave differently. For example, your equality relation could test only certain data members of two elements to determine element equality. In such cases, element equality may apply to two elements even when the elements are not exactly equal.

The equality relation for keys may be different than the equality relation for elements. Consider, for example, a job control block that has a priority and a job identifier that defines equality for jobs. You could choose to implement a job collection as unordered, with the job ID as key, or as sorted by priority, with the priority as key. The `Job` class for this job control block could look like this:

```
typedef unsigned long JobId;
typedef int Priority;
class Job {
     JobId ivId;          // These are private data members.
     Priority ivPriority;
public:
     JobId   id ()  const { return ivId; }
     Priority priority () { return ivPriority; }
};
// If ivId is the key:
JobId const& key (Job const& t)
{ return t.id (); }
// If ivPriority is the key:
Priority const& key (Job const& t)
{ return t.priority (); }
// ...
```

In the first case, you have fast access through the job ID but not through the priority; in the second case, you have fast access through the priority but not through the job ID. The ordering relation on the priority key in the second case does not yield a job equality, because two jobs can have equal priorities without being the same.

Functions like `locateElementWithKey()` (📖 described in "Flat Collection Member Functions" in the *Open Class Library Reference*) use the equality relation on keys to locate elements within a collection. A collection that defines key equality may also define element equality. Functions that are based on equality (such as `locate()`) are only provided for collections that define element equality. Collections that define neither key equality nor element equality, such as heaps and sequences, provide no functions for locating elements by their values or testing for containment. Elements can be added and retrieved from such collections by iteration. For sequences, elements can also be added and retrieved by position.

A sorted collection must define either key equality or element equality. A sorted collection that does not have a key defined must have an ordering relation defined for the element type. This relation implicitly defines element equality.

Keys can be used to access a particular element in a collection. The alternative to defining element equality as equality of all data members is to define it as equality of keys only. (In the job control block example on page 83, this means defining job equality as equality of the job ID.) Use this alternative only when you are sure that keys are unique. When you use this alternative, you can locate an element only with the key (using `locateElementWithKey(`*`key`*`)` instead of `locate(`*`element`*`)`. Locating elements by key improves performance, particularly if the complete element is large or difficult to construct in comparison to the key alone. Consider the two alternatives in the following example:

```
// First solution
JobId const& key (Job const& t) {  return t.id;  }
KeySet < Job, int > jobs;
// ...
jobs.locateElementWithKey (1);

// Second solution
IBoolean operator== (Job const& t1, Job const& t2)
{  return t1.id == t2.id;  }
Set < Job > jobs;
// ...
Job t1;
t1.id = 1;
jobs.locate (t1);
```

The first solution is superior, if job construction (`Job t1`) requires a significant proportion of the total system resources used by the program.

The Collection Class Library provides sorted and unsorted versions of maps and relations, for which both key and element equality must be defined. These collections are similar to key set and key bag, except that they define functions based on element equality, namely union and intersection. The `add()` function behaves differently toward maps and relations than it does toward key set and key bag.

Uniqueness of Entries

The terms *unique* and *multiple* relate to the key, in the case of collections with a key. For collections with no key, *unique* and *multiple* relate to the element.

In some flat collections, such as map, key set, and set, no two elements are equal or have equal keys. Such collections are called *unique collections*. Other collections, including relation, key bag, bag, and `HEAP`, can have two equal elements or elements with equal keys. Such collections are called *multiple collections*.

For those multiple collections with key that have element equality (relation and sorted relation), elements are always unique while keys can occur multiple times. In other words, if element equality is defined for a multiple collection with key, element equality is tested before inserting a new element.

A unique collection with no keys and no element equality is not provided because a *containment function* cannot be defined for such a collection. A containment function determines whether a collection contains a given element.

The behavior during element insertion (when one of the `add...` methods is applied to a collection) distinguishes unique and multiple collections. In unique collections, the `add()` function does not add an element that is equal to an element that is already in the collection. In multiple collections, the `add()` function adds elements regardless of whether they are equal to any existing elements or not.

The `add()` function has two general properties:

- All elements that are contained in the collection before an element is added are still contained in the collection after the element is added.
- The element that is added will be contained in the collection after it is added.

Operations that contradict these properties are not valid. You cannot add an element to a map or sorted map that has the same key as an element that is already contained in the collection, but is not equal to this element (as a whole). In the case of a map and sorted map, an exception is thrown. Note that both map and sorted map are unique collections. The functions `locateOrAddElementWithKey()` and `addOrReplaceElementWithKey()` specify what happens if you try to add an element to a collection that already contains an element with the same key.

Figure 8 on page 86 shows the result of adding a series of four elements to a map, a relation, a key set, and a key bag. The first row shows what each collection looks like after the element `<a,1>` has been added to each collection. Each following row shows what the collections look like after the element in the leftmost column is added to each.

The elements are pairs of a character and an integer. The character in the pair is the key. An element equality relation, if defined, holds between two elements if both the character and the integer in each pair are equal.

`add`	Map or sorted map	Relation or sorted relation	Key set or key sorted set	Key bag or key sorted bag
<a,1>	<a,1>	<a,1>	<a,1>	<a,1>
<b,1>	<a,1>, <b,1>	<a,1>, <b,1>	<a,1>, <b,1>	<a,1>, <b,1>
<a,1>	<a,1>, <b,1>	<a,1>, <b,1>	<a,1>, <b,1>	<a,1>, <b,1>, <a,1>
<a,2>	exception: Key Already Exists	<a,1>, <b,1>, <a,2>	<a,1>, <b,1>	<a,1>, <b,1>, <a,1>, <a,2>

Figure 8. Behavior of add for Unique and Multiple Collections

Restricted Access

Flat collections with restricted access have a restricted set of functions that can be applied to them; that is, only a subset of the functions listed in 🕮 "Flat Collection Member Functions" in the *Open Class Library Reference* can be applied. Examples of such flat collections are stack and priority queue.

You may want to restrict the set of functions for reasons such as:

1. You can simplify the interface to the collection.
2. The normal rules for restricted flat collections apply, so certain assumptions can be made when validating and inspecting the code. A stack, for example, does not allow the removal of any element except the top one.
3. You can create new implementation options.

The Collection Class Library provides the following flat collections with restricted access:

- Stack, deque, and queue, which are all based on sequence
- Priority queue, which is based on key sorted bag

🕮 See Part 3, "Flat Collection Classes" in the *Open Class Library Reference* for descriptions of collections with restricted access. These descriptions are alphabetically merged with descriptions for other collections. You can use Table 2 on page 87 to select the appropriate flat collection with restricted access for a given set of properties.

Table 2. Properties for Collections with Restricted Access

Add	Remove	Sorted (with key)	Unsorted (no key)
According to key	First	Priority queue	N/A
Last	Last	N/A	Stack
Last	First	N/A	Queue
First or last	First or last	N/A	Deque

Trees

Trees can be described either as structures where the elements have a hierarchy or as a special form of recursive structure. Recursively a tree can be described as a node (parent) with pointers to other nodes (children). Every node has a fixed number of pointers, which are set to null at initialization time. Insertion of a new node involves setting a pointer in the parent so that it points to the inserted child. Figure 9 illustrates the structure of an n-ary tree.

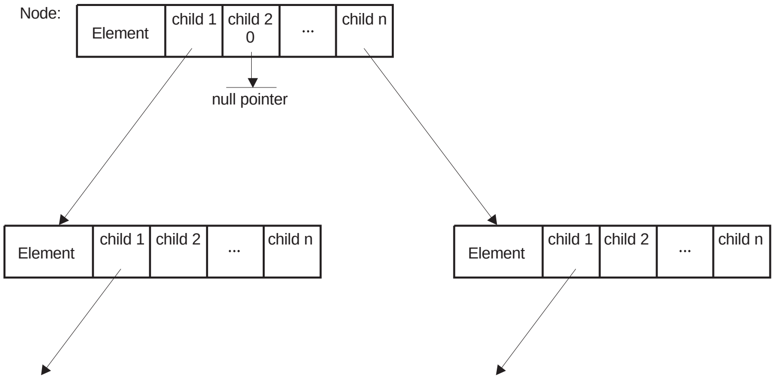


Figure 9. The Structure of N-ary Trees

Similarly, you can obtain tree-like or recursive structures by implementing the array of children of a node as a flat collection of nodes. This will give you different functionality for the children, for example, the ability to locate a child with a given value.

Auxiliary Classes

To use the collection classes, you also need a *cursor* and an *iterator* class. These are described in “Cursors” on page 98 and “Iteration Using Iterators” on page 102.

You can use the *pointer* and *managed pointer* classes to manage objects; they enable automatic storage management. “Using Pointer Classes” on page 115 and “Managed Pointers” on page 119 explain the concepts and usage in detail.

The Overall Implementation Structure

To achieve maximum runtime efficiency and ease of use, the Collection Class Library combines the common features of object-oriented techniques, such as class hierarchies, polymorphism and late binding, with an efficient class structure that uses advanced optimization techniques. This section gives a brief overview of the Collection Class structure that is shown in Figure 10 on page 90. A more detailed explanation of the particular concepts is found in subsequent sections.

You need not understand the entire implementation structure to begin using the collections in their basic forms. The following is a list of the implementation strategies offered by the Collection Class Library, in order of increasing complexity:

Use the Defaults

Default implementations are provided for every collection. If you do not want to be concerned with choosing an implementation for an abstract data type, you can use the *default classes* provided by the Collection Classes. In chapters of the *Open Class Library Reference* that describe particular collections, the default implementation is the first implementation in the “Class Implementation Variants” table for that chapter, if a table is present. If no table is present, the default implementation is stated in the chapter's “Class Implementation Variants” section.

Use Variants

If you want to choose a particular implementation variant for a collection, you can easily replace the default implementation by an implentation variant of the same collection that behaves externally in the same way but offers improved performance.

Use Polymorphism

If you want to have a more generalized collection class than those offered by the concrete classes, you can take advantage of polymorphism. For example, when working with a set, instead of using the concrete classes `ISet, IGSet, ISetOnBSTKeySortedSet`, and so on, you can use the abstract class `IASet` or, for more generic behavior, the abstract class `IAEqualityCollection`. *Abstract classes*, which are accessed using *reference classes*, let you program to a more generalized interface, without necessarily knowing what abstract data types (collections) your code will operate on. You can leave the implementation details for later.

Categories of Classes

The hierarchy of abstract classes lets you overload selected collection class member functions. You can inherit from an appropriate reference class, and then implement the member functions that you want to overload. For all other member functions, the reference class calls the corresponding methods from the concrete ("based-on") class hierarchy. You can also use the reference classes to achieve polymorphism, which is discussed on page 139.

Figure 10 on page 90 illustrates the relationships between the categories of classes for the collection known as a set. Each class falls within one of five categories: concrete, typed implementation, typeless implementation, abstract, and reference classes. Arrows indicate a relationship between classes. Text beside each arrow indicates the relationship between the two classes. The relationships are:

- Instantiates
- Is a
- Uses

In this figure, you will notice certain naming conventions. For example, default classes begin with the letter `I`, while abstract classes begin with the letters `IA`. For information on naming conventions, see "Class Template Naming Conventions" on page 93.

Implementation Structure

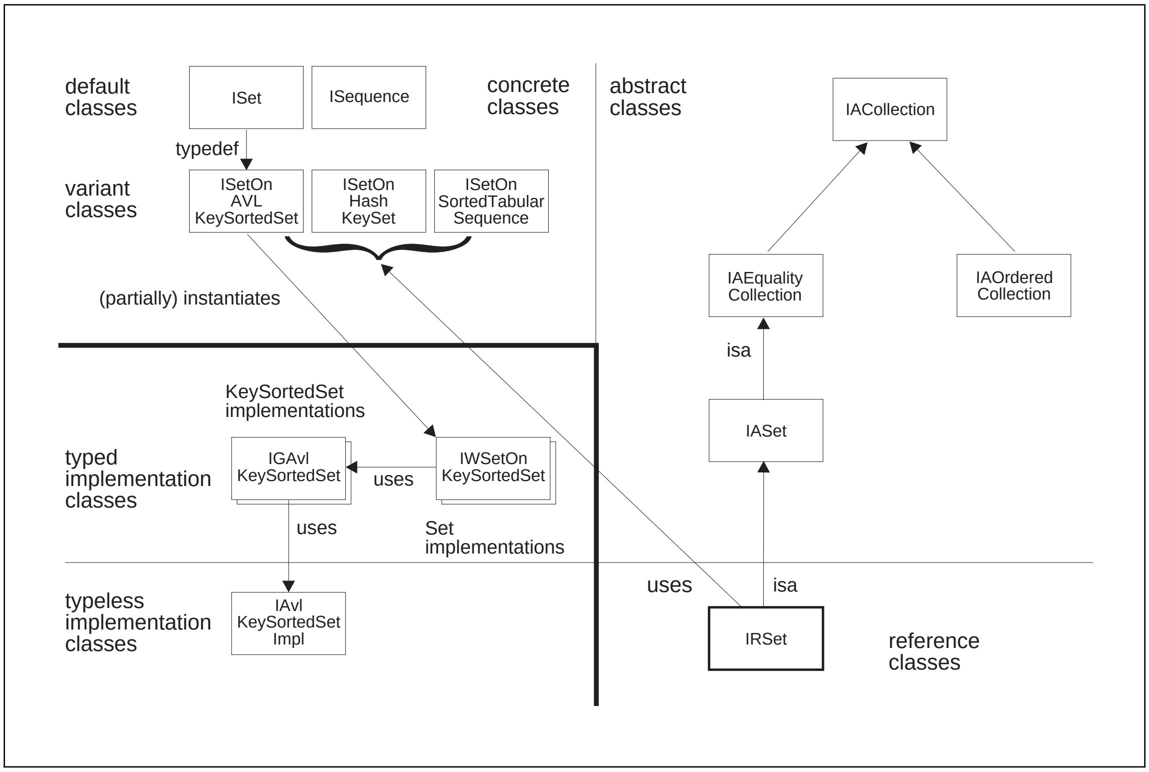


Figure 10. Overall Library Structure

The following sections describe the categories of collections in the Collection Class Library.

Default Classes

The *default* classes provide the easiest way to use the collection classes. Two default classes are provided for each abstract data type:

- A class that is instantiated only with the element type, and possibly the key type. `ISet` is an example of this type of default class.
- A class that takes element-specific functions. `IGAvlKeySortedSet` is an example of this type of default class. 🕮 See “Using Element Operation Classes” on page 110 for information on element-specific functions.

Variant Classes

Each abstract data type can be instantiated either by its default class or by one of several variant classes. Sets can be implemented, for example, as key sorted sets or as hash tables. Key sorted sets, in turn, can be implemented as linked or tabular or diluted sequences. Default classes and variant classes are also called the implementation variants of a collection. All implementation variants of a collection have the same interface and external behavior.

Abstract Classes

The classes in the Collection Classes are all related through the hierarchy of *abstract* classes shown in Figure 11 on page 92. In the figure, abstract classes have a grey shadow. Concrete collections have a black shadow, or a white shadow for restricted access collections. The leaves of the abstract class hierarchy (that is, those classes that have no derived classes within the abstract class hierarchy tree) define the collection for which concrete implementations are provided. The lines in the figure represent an *is a* relationship from a lower collection to the collection above it. For example, a set is an equality collection, which is a collection. Note that Tree does not inherit from any abstract class. The names of abstract collections start with `IA`.

Reference Classes

To avoid the overhead of virtual function calls, the Collection Classes do not allow concrete classes to be derived directly from abstract classes. The compiler can usually optimize function calls when it knows the exact type of the object, but because collections are mostly passed by reference, such an optimization is not possible with the collection classes. If you do not want to take advantage of polymorphism, you do not have to deal with the overhead of virtual function calls.

Abstract and concrete classes are linked through *reference* classes. These classes are derived from the abstract classes, and implement the member functions using one of the corresponding concrete classes. Names of reference classes start with `IR`. 📖 See Chapter 11, “Polymorphic Use of Collections” on page 139 for more details on the use of polymorphism in the Collection Classes.

Support Classes for Visual Builder for C++

The collection classes have special classes which support Visual Builder for C++. See *Visual Builder User's Guide* for more information on Visual Builder.

Typed and Typeless Implementation Classes

Typed implementation classes implement the concrete classes. They provide an interface that is specific to a given element type.

Typed implementation classes may be basic or based on another implementation. Basic classes have names that start with `I` or `IG`. Based-on classes have names that start with `IW`. 📖 For further details, see “The Based-On Concept” on page 126.

Typeless implementation classes prevent unnecessary code expansion, which could occur if all code for a collection were fully implemented through its templates. For example, the add(Element const& *element*) function is offered with a typed interface,

Implementation Structure

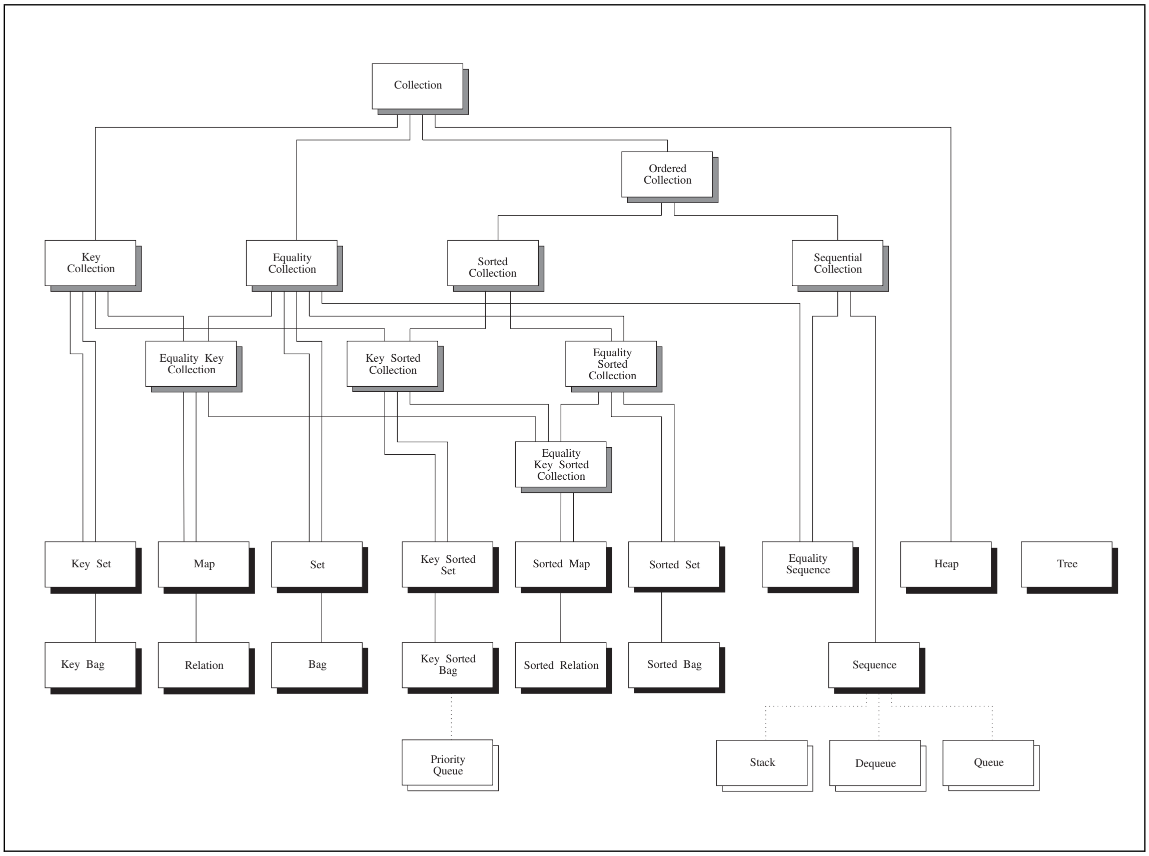


Figure 11. The Abstract Class Hierarchy. Abstract classes have a grey background. Concrete classes have a black background. Restricted access classes have a white background. Dotted lines show a "based-on" relationship, not an actual derivation.

so that the compiler can check whether a program tries to add a string to a collection of integers. However, suppose an application were to use all of the following:

```
integerCollection.add(anInteger);
stringCollection.add(aString);
elementCollection.add(anElement);
//...
```

Without typeless implementations, each collection's template instantiation of the `add()` function would need to contain the full functionality for adding an element. By having each of these typed `add()` functions use the same typeless (**`void*`**)implementation code, the library avoids unnecessary code expansion.

The collection classes, however, use functions that return specific types. The implementation classes provide an untyped (`void*`) interface that the concrete class implementations use.

Class Template Naming Conventions

All class templates begin with an uppercase `I`. Table 3 shows the naming conventions used to distinguish between different types of class templates, given a default class template of `ISet`. Underscored letters in each class template name are those that indicate the stated convention:

Class name	Meaning of letters
`ISet`	Default class template.
`IGSet`	Default generic class template. The element operations class can be specified as template argument.
`ISetOn...`	Variant class template.
`IVSet`	Support class template for Visual Builder
`IVGSetOn`	Generic support class template for Visual Builder. The element operations class can be specified as a template argument.
`IWSetOn...`	Typed implementation based on another typed implementation. You can think of the `W` as a shorthand for "wrapping another implementation with a new interface." (See "Typed and Typeless Implementation Classes" on page 91 for further details.)
`IASet`	Abstract class template.
`IRSet`	Reference class template.

Table 3. Class Template Naming Conventions

Linking to the Collection Classes

The Collection Class Library uses the library files shown below. By default the compiler uses dynamic linking, and you should not have to specify any library. The files are shown here in case you want to override the default behavior:

- `CPPOOC30.LIB` - for static linking
- `CPPOOC3I.LIB` - import library for dynamic linking
- `CPPOOC30.DLL` - for dynamic linking

Linking to the Collection Class Library

8 Instantiating and Using the Collection Classes

This chapter describes how to instantiate and use collection classes.

To use a collection class, you normally follow these three steps:

1. Instantiate a collection class template and provide arguments for the formal template arguments.
2. Define one or more objects of this instantiated class, possibly providing constructor arguments.
3. Apply functions to these objects.

Instantiation and Object Definition

This section describes instantiation for the default implementation. For a given class, such as `ISet`, and a given element type, such as a class named `Job`, the instantiation for a new class that represents sets of jobs could look like this:

```
typedef ISet < Job > JobSet;
```

The instantiation could also look like this:

```
class JobSet : public ISet < Job > {
public:
     JobSet (INumber n = 100) : ISet < Job > (n) {}
};
```

The second form defines a new class called `JobSet` that has a constructor that takes a single argument. The definition of the constructor is necessary if the program needs to create `JobSets` with different estimates for the number of elements. Because derived classes do not inherit their constructors from their base classes, `JobSet` does not inherit the constructor of `ISet < Job >`.

Once the `JobSet` collection is defined, you can define `JobSet` objects `toBeDone`, `pending`, and `delayed` as follows:

```
JobSet toBeDone, pending, delayed;
```

You can also define the objects without introducing a new type name (`JobSet`):

```
ISet < Job > toBeDone, pending, delayed;
```

However, you should begin by explicitly defining a named class, such as `JobSet`, that uses the default implementation. It is then easier to replace the default implementation with a better implementation later on. See Chapter 10, “Tailoring a Collection Implementation” on page 125 for more details on replacing default implementations.

Bounded and Unbounded Collections

A *bounded* collection limits the number of elements it can contain. There are no bounded collections in the Collection Classes. The concept of bounded collections is supported so that you can create your own bounded collection implementations.

When a bounded collection contains the maximum number of elements (its bound), the collection is said to be full. This condition can be tested by the function `isFull()`. If elements are added to a full collection, the exception `IFullException` is thrown. This behavior is useful for collections that are to have their storage allocated completely on the runtime stack.

You can determine the maximum number of elements in a bounded collection by calling the function `maxNumberOfElements()`. You can only call this function if the collection is bounded. You can determine whether a collection is bounded by calling the function `isBounded()`.

In the current implementation of the Collection Classes, all collections are unbounded. The functions `isBounded()` and `isFull()` always return `False`. The function `maxNumberOfElements()` throws the exception `INotBoundedException`.

Adding, Removing, and Replacing Elements

You can perform three operations to modify a collection:

- Adding elements. Use the `add()` function and its variants.
- Removing elements. Use the `remove()` function and its variants.
- Replacing elements. Use the `replace()` function and its variants.

Adding Elements

The function `add()` places the element identified by its argument into the collection. After an element has been added, all cursors of the collection become undefined. Cursors are used to point to elements of the collection; an undefined cursor is one that might not currently point to a valid element. `add()` behaves differently depending on the properties of the collection:

- In unique collections, an element is not added if it is already contained in the collection.
- In sorted collections, an element is added according to the ordering relation of the collection.
- In sequential collections, an element is added to the end of the collection.

In general, you can copy one collection to another collection that is initially empty by iterating through the elements of the first collection and calling `add()` with each element as an argument. In particular, for a sequential collection, `add()` must add the element last, because iteration iterates from the first toward the last element.

For sequential collections, elements can be added at a given position using add functions other than `add()`, such as `addAtPosition()`, `addAsFirst()`, and `addAsNext()`. Elements after and including the given position are shifted. Positions can be specified by a number, with 1 for the first element, by using the `addAtPosition()` function. Positions can also be specified relative to another element by using the `addAsNext()` or `addAsPrevious()` functions, or relative to the collection as a whole by using the `addAsFirst()` or `addAsLast` functions.

Removing Elements

In the Collection Classes, you can remove an element that is pointed to by a given cursor by using the `removeAt()` function. All other removal functions operate on the model of first generating a cursor that refers to the desired position and then removing the element to which the cursor refers. There is an important difference between element *values* and element *occurrences*. An element value may, for nonunique collections, occur more than once. The basic `remove()` function always removes only one occurrence of an element.

For collections with key equality or element equality, removal functions remove one or all occurrences of a given key or element. These functions include `remove()`, `removeElementWithKey()`, `removeAllOccurrences()`, and `removeAllElementsWithKey()`. Ordered collections provide functions for removing an element at a given numbered position. Ordered collections also allow you to remove the first or last element of a collection using the `removeFirst()` or `removeLast()` functions.

After an element has been removed, all cursors of the collection become undefined. Therefore, removing all elements with a given property from a collection cannot be done efficiently using cursors. After you have removed one element with the property, the entire collection would have to be searched for the next element with the property. If you want to remove all of the elements in a collection that have a given property, you should use the function `removeAll()` and provide a *predicate* function as its argument. This predicate function has an element as argument and returns an `IBoolean` value. The `IBoolean` result tells whether the element will be removed. The following example removes all even elements from an integer collection:

```
IBoolean isEven (int const& i, void*)
{
    return i % 2 == 0;
}
// ...
  intSet.removeAll (isEven);
```

Sometimes you may want to pass more information to the predicate function. You can use an additional argument of type `void*`. The pointer then can be used to access a structure containing further information. 📖 See the last example under "Iteration Using Iterators" on page 102 for information on how to use the additional argument.

Replacing Elements

It is possible to modify collections by replacing the value of an element occurrence. Adding and removing elements usually changes the internal structure of the collection. Replacing an element leaves the internal structure unchanged. If an element of a collection is replaced, the cursors in the collection do not become undefined.

For collections that are organized according to element properties, such as an ordering relation or a hash function, the replace function must not change this element property. For key collections, the new key must be equal to the key that is replaced. For nonkey collections with element equality, the new element must be equal to the old element as defined by the element equality relation. The key or element value that must be preserved is called the *positioning property* of the element in the given collection type.

Sequential collections and heaps do not have a positioning property. Element values in sequences and heaps can be changed freely. Replacing element values involves copying the whole value. If only a small part of the element is to be changed, it is more efficient to use the `elementAt()` access function described in "Using Cursors for Locating and Accessing Elements" on page 100. The `replaceAt()` function checks whether the replacing element has the same positioning property as the replaced element. (📖 See Chapter 13, "Exception Handling" on page 145 for more details on preconditions.) When you use the `elementAt()` function to replace part of the element value, this check is not performed. If you want to ensure safe replacement (a replacement that does not change the positioning property), use `replaceAt()` rather than `elementAt()`.

Cursors

A *cursor* is a reference to an element in a collection. If the position of the element changes, the cursor is invalidated. This occurs because the cursor refers only to the position of the element and not to the element itself.

Cursors

A cursor is always associated with a collection. The collection is specified when the cursor is created. Each collection function that takes a cursor argument has a precondition that the cursor actually belong to the collection. Simple functions, such as advancing the cursor, are also functions of the cursor itself. For example, given the following definitions:

```
typedef ISet<Job> JobSet;
JobSet myJobSet;
JobSet::Cursor myCursor(myJobSet);
```

the following two lines of code are functionally equivalent:

```
myCursor.setToNext();
myJobSet.setToNext(myCursor);
```

Cursors and iteration by cursors can be used with any collection. With cursors the Collection Classes provide:

- An iteration scheme that is simpler than using iterators. (See “Iteration Using Iterators” on page 102.)
- The ability to define functions that return cursors. Such functions can give you fast access to an element if it exists, or indicate the non-existence of an element by returning an invalid cursor.

Cursors are only temporarily defined. As soon as elements are added to or removed from the collection, existing cursors become undefined. One of the three following situations occurs:

1. The cursor is invalidated (`isValid()` will return `False`).
2. The cursor remains valid and points to an element of the collection; however, it may point to a different element than before.
3. The cursor remains valid but no longer points to an element of the collection.

Do not use an undefined cursor as an argument to a function that requires the cursor to point to an element of the collection.

Each **concrete** collection class, such as `ISet<int>`, has an inner definition of a class `Cursor` that can be accessed as `ISet<int>::Cursor`.

Because **abstract** classes declare functions on cursors just as concrete classes do, there is a base class `ICursor` for these specific cursor classes. To allow the creation of specific cursors for all kinds of collections, every abstract class has a virtual member function `newCursor()`. `newCursor()` creates an appropriate cursor for the given collection object.

Cursors

Using Cursors for Locating and Accessing Elements

Cursors provide a basic mechanism for accessing elements of collection classes. For each collection, you can define one or more cursors, and you can use these cursors to access elements. Collection Class functions such as `elementAt()`, `locate()` and `removeAt()` use cursors.

`elementAt()` lets you access an element using a cursor.

`elementAt()` returns a reference to an element, thereby avoiding copying the elements. Suppose that an element had a size of 20KB and you want to access a 2-byte data member of that element. If you use `elementAt()` to return a reference to this element, you avoid having to copy the entire element to a local variable.

Several other functions, such as `firstElement()` or `elementWithKey()`, return a reference to an element. They can be thought of as first executing a corresponding cursor function, such as `setToFirst()` or `locateElementWithKey()`, and then accessing the element using the cursor.

You must determine if the element exists before trying to access it. If its existence is not known from the context, it must first be checked. To save the extra effort of locating the desired element twice (once for checking whether it exists and then for actually retrieving its reference), use the cursor that is returned by the locate function for fast element access:

```
if (myCollection.locateElementWithKey (someKey, myCursor)) {
     // ...
     myVariable = myCollection.elementAt (myCursor);
     // ...
  }
```

The `elementAt()` function can also be used to replace the value of the referenced element. You must ensure that the positioning property of the element is not changed with respect to the given collection. 📖 See “Adding, Removing, and Replacing Elements” on page 96 for more details.

There are two versions of `elementAt()`:

```
Element const& elementAt (ICursor const&) const;
Element&       elementAt (ICursor const&);
```

Use the first version of `elementAt()` if you want to assert to the compiler that no elements in the collection can be changed by this function.

Iterating over Collections

Iterating over all or some elements of a collection is a common operation. The Collection Class Library gives you two methods of iteration:

- Using cursors
- Using iterators or iteration functions

Ordered (including sorted) collections have a well-defined ordering of their elements, while unordered collections have no defined order in which the elements are visited in an iteration. Each element is visited exactly once.

You cannot add or remove elements from a collection while you are iterating over a collection, or all elements may not be visited once. You cannot use any of the iterations described in this section if you want to remove all of the elements of a collection that have a certain property. Use the function `removeAll()` (described in "Flat Collection Member Functions" in the *Open Class Library Reference*), that takes a predicate function as argument. See "Removing Elements" on page 97 for details on removing elements.

Iteration Using Cursors

Cursor iteration can be done with a **`for`** loop. Consider the following example:

```
ISet<int> myCollection;
ISet<int>::Cursor myCursor (myCollection);
for (myCursor.setToFirst (); myCursor.isValid ();
     myCursor.setToNext ())
{
     // ...
     int currentElement = myCollection.elementAt (myCursor) ;
     // change currentElement
     // ...
}
```

`ISet<int>::Cursor` is the class `Cursor` that is defined within the class `ISet<int>`. This is referred to as a *nested class*. `myCursor` is the name of the cursor object. Its constructor takes `myCollection` as argument.

The Collection Class Library defines a macro `forCursor` that you can use to write an elegant cursor iteration:

```
 #define forCursor(c)         \
   for ((c).setToFirst();  \
        (c).isValid();     \
        (c).setToNext())

// myCollection and myCursor are the same as before.

forCursor(myCursor)
{
     // ...
     int currentElement = myCollection.elementAt (myCursor);
     // change currentElement
     // ...
}
```

If the element is used as read-only, a function of the cursor can be used instead of `elementAt(myCursor)`:

```
// myCollection and myCursor are the same as before.
// myCursor's construction associated it to myCollection.

forCursor(myCursor)
{
     // ...
     int currentElement = myCursor.element ();
     // print currentElement
}
```

The function `element()` above is a function of the Cursor class (see “Cursors” on page 98). It returns a **`const`** reference to the element currently pointed at by the cursor.

Note: You should remove multiple elements from a collection using the `removeAll()` function, with a predicate function as an argument. This function is more efficient and less error-prone than the alternative of removing elements using cursor iteration. See “Adding, Removing, and Replacing Elements” on page 96 for further details.

Iteration Using Iterators

Cursor iteration has two possible drawbacks:

- For unordered collections, the explicit notion of an (arbitrary) ordering may be undesirable for stylistic reasons. For example, it could mislead you (or another programmer) into perceiving or exploiting an order where in fact the order does not exist or is not guaranteed.
- Iteration in an arbitrary order might be done more efficiently using something other than cursors. For example, with tree representations, a recursive descent iteration may be faster than the cursor navigation, even though the time for extra function calls must be considered.

Iteration

The Collection Class Library provides the allElementsDo() function that addresses both drawbacks by calling a function that is applied to all elements. The function returns an IBoolean value that tells whether the iteration should be continued or not. For ordered collections, the function is applied in the order of elements within the collection. Otherwise the order is unspecified.

The function that is applied in each iteration step can be given in two ways: directly as a C++ function, or by defining the function as a method of a user-defined iterator class:

- **As a C++ function**: Code the function that you want to be applied to all elements as a C++ function, then use allElementsDo() to apply the function to the elements.
- **As an object of an iterator class**: Code the function as a member function of an iterator class that you create (for example, *myIteratorClass*). Then let the iterator apply this function to every element, by using allElementsDo(*myIteratorObject*), where *myIteratorObject* is an object of *myIteratorClass*.

The second possibility is more flexible. You can better encapsulate the member function, and you can use additional arguments to that function if needed. If the function is a method that you can use for various classes, you can reuse the iteration class.

Note: Do not add or remove elements while using the iterator.

For both these possibilities (the C++ function and the object of an iterator class), an additional distinction is made as to whether the function leaves the element constant or not. This means that four definitions of the function allElementsDo() are offered by every collection. The following example shows the definition of allElementsDo() for ISet:

```
   template < class Element, ... >
   class ISet {
      // ...
                     // Iteration applying a C++ function:
      IBoolean allElementsDo (IBoolean (*function)(Element&, void*),
                          void* additionalArgument = 0);
      IBoolean allElementsDo (IBoolean (*function)(Element const&, void*),
                          void* additionalArgument = 0) const;

                     // Iteration applying an iterator object:
      IBoolean allElementsDo (IIterator < Element > &);
      IBoolean allElementsDo (IConstantIterator < Element > &)const;
};
```

Iteration

If you use an object of an iterator class, this class must offer an applyTo() function. It also must be derived from the abstract base class IIterator or IConstantIterator. These abstract iterator base classes are defined in the following way:

```
template < class Element >
class IIterator {
public:
      virtual IBoolean applyTo (Element&) = 0;
};

template < class Element >
class IConstantIterator {
public:
      virtual IBoolean applyTo (Element const&) = 0;
};
```

Additional arguments that are needed for the iteration can, for example, be passed as arguments to the constructor of the derived iterator class. You must define the function with the given argument and return types. For additional arguments, you may have to define a separate class or structure.

The following example shows the use of iterators. The example adds all integers in a bag using two methods: by iterating the applied function as an object of an iterator class or as a function.

```
//    An example of using Iterators

#include <ibag.h>
#include <iostream.h>

typedef IBag < int > IntBag;

class SumIterator : public IConstantIterator < int > {
     int ivSum;
public:
      SumIterator () : ivSum (0) {}
      IBoolean applyTo (int const& i) {
           ivSum += i;              // Increments ivSum by the value
           return True;             // of the current element
      }
      int sum () { return ivSum; }  // used to return the sum of
                                    // integers in the bag as
                                    // calculated by applyTo
};

int sumUsingIteratorObject (IntBag const& bag) {
     SumIterator sumUp;             // Instantiates an iterator object
     bag.allElementsDo (sumUp);     // of SumIterator in order to
     return sumUp.sum ();           // apply its methods to the bag
}                                   // of integers

IBoolean sumUpFunction (int const& i, void* sum) {
    *(int*)sum += i;                // Increments sum by current value
    return True;                    // of i.  This function is applied
}                                   // to all elements of the bag.
```

```
int sumUsingIteratorFunction (IntBag const& bag) {
    int sum = 0;                    // Applies sumUpFunction (an
    bag.allElementsDo               // iterator function) to all
       (sumUpFunction, &sum);       // elements in the bag.
    return sum;
}

int main (int argc, char* argv[])  {
    IntBag intbag;
    for (int cnt=1; cnt < argc; cnt++)
       intbag.add(atoi(argv[cnt]));
    cout << "Sum obtained using an Iterator Object = "
         <<  sumUsingIteratorObject(intbag)  << "\n";
    cout << "Sum obtained using an Iterator Function = "
         <<  sumUsingIteratorFunction(intbag)  << "\n";
    return 0;
}
```

If you invoke this program by entering:

```
sumup 1 2 3 4 5
```

the program produces the following output:

```
Sum obtained using an Iterator Object = 15
Sum obtained using an Iterator Function = 15
```

Copying and Referencing Collections

The Collection Classes implement no structure sharing between different collection objects. The assignment operator and the copy constructor for collections are defined to copy all elements of the given collection into the assigned or constructed collection. You should remember this point if you are using collection types as arguments to functions. If the argument type is not a reference or pointer type, the collection is passed by the copy constructor, and changes made to the collection within the called function do not affect the collection in the calling function.

If you want a function to modify a collection, pass the collection as a reference:

```
void removePrimes (ISet < int > mySet) { /* ... */ }  // wrong
void removePrimes (ISet < int >& mySet) { /* ... */ } // right
```

For the sake of efficiency, avoid having a collection type as the return type of a function:

```
ISet < int > f () {
     ISet < int > result;
     // ...
     return result;
}
// ...
intSet = f ();   // inefficient
```

Copying and Referencing Collections

In this program `intSet` becomes a reference argument to the assignment operation, which would again copy the set. A better approach is:

```
void f (ISet < int > &result) { /* ... */ }
   // ...
f (intSet);
```

9

Element Functions and Key-Type Functions

This chapter describes the functions that are required by member functions of the Collection Classes to manipulate elements and keys. The following topics are discussed:

- Element functions and key-type functions
- Using standard operators to provide element and key-type functions
- Using separate functions
- Using element operation classes
- Functions for derived element classes

Introduction to Element Functions and Key-Type Functions

The member functions of the Collection Class Library call other functions to manipulate elements and keys. These functions are called *element functions* and *key-type functions*, respectively.

Member functions of the Collection Class Library may, for example, use the element's assignment or copy constructors for adding an element, or they may use the element's equality operator for locating an element in the collection. In addition, Collection Class functions use memory management functions for the allocation and deallocation of dynamically created internal objects (such as nodes in a tree or a linked list).

The element functions that may be required by a given collection are:

- Default and copy constructor
- Destructor
- Assignment
- Equality test
- Ordering relation
- Key access
- Hash function

The key-type functions that may be required by a given collection are:

- Equality test
- Ordering relation
- Hash function

Note: For implementation variants where both equality test and ordering relation are required element functions (or where both are required key-type functions), the library does not define which of the two is used to determine element or key equality.

The memory management functions that may be required by a given collection are:

- Allocation
- Deallocation

The lists above are the superset of all element functions and key-type functions that a Collection Class can ever require. For example, a collection without keys does not require any key-type functions, and a collection without element equality does not require an equality test. Element functions and key-type functions required for a certain collection are listed with the description of each collection in the *Open Class Library Reference*.

Where possible, these functions are already defined by the Collection Class Library. Default memory management functions are provided for usage with any element and key type. For the standard C++ data types `int` and `char*`, defaults are offered for all element and key-type functions. For all other element and key types, you must provide these functions.

There are three different methods of providing element functions and key-type functions, each of which offers a different level of flexibility and tailoring:

1. Using member functions
2. Using separate functions in the global name space
3. Using element operation classes.

The second and third methods can also be used to replace the default memory management functions for some of the collections.

Using Member Functions

The easiest way to provide the required element or key-type functions is to use member functions. For assignment, equality, and ordering relation, `operator=`, `operator==`, and `operator<` are used, respectively. Certain element functions and key-type functions must be defined as member functions. Others cannot be defined as member functions, but must be defined as separate functions.

You must define these functions using member functions:

- Constructors
- Destructors

You cannot define these functions using member functions. Instead you must define them as separate functions that are not members of any class:

- Functions for key access
- Functions for hashing
- Functions for memory management

Except for assignment, you must define member functions of a class as **const**. You will get a compile-time error if you do not include **const** in these definitions.

The following example shows how member functions must be defined as **const**:

```
class Element
{
public:
     Element&       operator=  (Element const&);
     IBoolean       operator== (Element const&) const;
     IBoolean       operator<  (Element const&) const;
};
```

The result type of the assignment operator is irrelevant to the Collection Class Library. The result type of equality and ordering relation must be compatible with type `IBoolean`.

Using Separate Functions

You can use separate functions to provide the required element and key functions. Use separate functions when, in instantiating the Collection Class, you have no control over the element class, and the element class does not define the appropriate functions. You can also use separate functions to provide key access and hash function.

The following shows what the declarations for these separate functions must look like:

```
void          assign  (Element&, Element const&);
IBoolean      equal   (Element const&, Element const&);
long          compare (Element const&, Element const&);
Key const&    key     (Element const&);
unsigned long hash    (Element const&, unsigned long);
IBoolean      equal   (Key const&, Key const&);
long          compare (Key const&, Key const&);
unsigned long hash    (Key const&, unsigned long);
```

You can find examples of these functions in the tutorials (see Chapter 14, “Collection Class Library Tutorials” on page 151) and in the coding examples in the *Open Class Library Reference*.

You can also use separate functions for the standard memory management functions, as defined by the C++ language:

```
void*         operator new (size_t);
void          operator delete (void*);
```

The `compare()` function must return a value that is less than, equal to, or greater than zero, depending on whether the first argument is less than, equal to, or greater than the second argument. The hash function must return a value that is less than the second argument; this value may be achieved, for example, by computing the remainder (`operator%`) with the second argument. The hash function should evenly distribute over the range between zero and the second argument. For equal elements or keys, the hash element must yield equal results.

An efficient hash function is very important to the performance of your program. If you are unsure of how to implement an efficient hash function, see the suggested reading material on data structures and algorithms in “Other Books You Might Need” on page 716.

For assign(), equal(), and compare(), template functions are defined that will be instantiated unless otherwise defined. The default for assign() uses the assignment operator, the default for equal() uses the equality operator, and the default for compare() uses two comparisons with operator<. It is therefore advisable to define your own compare() function if the given element type has a more efficient implementation available. Such definitions are already provided for integer types using operator- and for char* using strcmp(). By default, the standard memory management functions are used. (Using operator- works for integer types because the result of a-b can be used to determine whether a<b evaluates to True.)

The following example demonstrates the use of a separate function for the definition of the key access. The element class is Task, its data member ivId is the key, and its member function id() is used to access the key:

```
typedef unsigned long TaskId;
typedef int Priority;
class Task {
     TaskId ivId;          // This will be used as the key.
     Priority ivPriority;   // These are private data members.
public:
     TaskId   id ()  const { return ivId; }
     Priority priority () { return ivPriority; }
    // ... other member functions, for example one that
    //     sets or changes a task's priority
};
// ...
TaskId const& key (Task const& t)    // Key access.
{ return t.id (); }
// The key() function cannot directly return the key (ivId)
// because the key is a private data member.
// ...
IKeySet <Task, TaskId> runningTasks;
```

Using Element Operation Classes

You can use element operation classes in cases where you want to place elements of one type into more than one collection, and where the element or key-type functions are different for each collection. For example, suppose you require an element type that is used to instantiate employee records that can be sorted either by name or by salary. You can declare an element class Person, and then place references to each Person instance into each of two collections. In one collection, the key is the name; in the other, the key is the salary. In your program, you need to define different element and key-type functions for hashing, comparison, and so on. Because these functions are not identical for both collections, you cannot define them within the class Person.

You can provide different sets of element and key-type functions for a given element type and multiple collections, by using the IG... class template for the collection you want to use. This class template lets you define element functions separately from the element class. In the case of the employee program, you can declare two classes as follows:

```
IGKeySortedSet <PersonPtr, int, SalaryOps> SalaryKSet;
IGKeySortedSet <PersonPtr, IString, NameOps> NameKSet;
```

You then need to define two other classes, SalaryOps and NameOps, which must contain appropriate element and key-type functions.

When you do not provide element or key operations by using an IG... collection, the standard class template (I... as opposed to IG...) defines default operations. These default operations are declared in istdops.h.

For an example of using element operation classes, see "Coding Example for Map" in the *Open Class Library Reference*.

The following excerpt shows the definition of the class templates for ILinkedSequence and IGLinkedSequence:

```
template < class Element, class ElementOps >
class IGLinkedSequence { /* ... */ };

template < class Element >
class ILinkedSequence :
  public IGLinkedSequence < Element, IStdOps < Element > > {
          /* ... */ };
```

The advantage of passing the arguments using an extra class instead of passing them as function pointers is that the class solution allows inlining.

The following is a skeleton for operation classes. The keyOps member must only be present for key collections. Note that all element and key operations must be defined as **const**.

```
template < class Element, class Key >
class ...Ops
{
     void*         allocate     (size_t) const;
     void          deallocate   (void*) const;
     void          assign       (Element&, Element const&) const;

     IBoolean      equal        (Element const&, Element const&) const;
     long          compare      (Element const&, Element const&) const;
     Key const&    key          (Element const&) const;
     unsigned long hash         (Element const&, unsigned long) const;
     class KeyOps
     {
          IBoolean      equal         (Key const&, Key const&) const;
          long          compare       (Key const&, Key const&) const;
          unsigned long hash          (Key const&, unsigned long) const;
     }
     keyOps;
};
```

You can inherit from the following class templates when you define your own operation classes. Templates with argument type `T` can be used for both the element and the key type.

```
class IStdMemOps
{
     void* allocate (size_t) const;
     void deallocate (void*) const;
};

template < class T >
class IStdAsOps
{
     void assign (T&, T const&) const;
};

template < class T >
class IStdEqOps
{
     IBoolean equal (T const&, T const&) const;
};

template < class T >
class IStdCmpOps
{
     long compare (T const&, T const&) const;
};

template < class Element, class Key >
class IStdKeyOps
{
     Key const& key (Element const&) const;
};

template < class T >
class IStdHshOps
{
     unsigned long hash (T const&, unsigned long) const;
};
```

The file `istdops.h` defines the above templates. It also defines other templates that combine the properties of one or more of the templates. The following table shows all template class names defined in `istdops.h`, and the element and key-type functions they implement:

Template	allocate deallocate	assign	equal	compare	hash	key using compare	key using equality and hash
IStdMemOps	√						
IStdAsOps		√					
IStdEqOps			√				
IStdCmpOps				√			
IStdHshOps					√		
IStdOps	√	√					
IEOps	√	√	√				
IECOps	√	√	√	√			
IEHOps	√	√	√		√		
IKCOps	√	√				√	
IKEHOps	√	√					√
IEKCOps	√	√	√			√	
IEKEHOps	√	√	√				√

To define an operations class, use the predefined templates for standard functions, and define the specific functions individually. Consider, for example, tasks that have an identifier and a priority. The identifier might serve as the key in a collection that keeps track of all active tasks, while the priority might be used for implementing priority-controlled task queues. Because the `key()` function is already defined to yield the task identifier, the priority queue has to be instantiated in the following way:

```
class TaskPrioOps : public IStdMemOps,
                    public IStdAsOps < Task >
{
public:
     Priority key (Task const& t)  { return t.priority (); }
     IStdCmpOps < Priority > keyOps;
};
// ...
IGPriorityQueue < Task, Priority, TaskPrioOps >
       taskPriorityQueue;
```

The functions that are required for a particular Collection Class depend not only on the abstract class but also on the concrete implementation choice. If you choose a set to be implemented through a hash table, the elements require a hash function. If you choose a (sorted) AVL tree implementation, elements need a comparison function. Even the default implementations may require more functions to be provided than would be necessary for the collection interface. Each chapter in the *Open Class Library Reference* that describes a particular collection defines which functions must be provided for keys and elements for each implementation of that collection.

Memory Management with Element Operation Classes

The following scenario illustrates the use of memory management with element operation classes.

Suppose you want to use your own element operation class to provide a special form of memory management. For example, you want an entire collection (the collection body plus the elements) to reside in a database, or in shared memory. To do this you can code:

```
IGLinkedSequence<Element, MyOperationsClass>
```

where MyOperationsClass is an element operations class you have coded, which provides your own element operations allocate() and deallocate(). This class may or may not inherit from previously described template classes, except that it must inherit from IStdMemOps).

A certain instance of your collection is instantiated together with an instance of your MyOperationsClass. You can retrieve the **this** pointer of this instance of MyOperationsClass to find out where the collection is instantiated, and you can use this address in your implementation of the allocate() element function to allocate your elements in the same memory pool where your collection resides.

Functions for Derived Element Classes

One of the C++ language rules states that function template instantiations are considered before conversions. Because the Collection Class Library defines default templates for element functions, functions such as equal() or compare(), defined for a class, will not be considered for that class's derived classes; the default template functions will be instantiated instead. In the following example, the compiler would attempt to instantiate the template compare() function for class B, instead of inheriting the compare() function of class A and converting B to A:

```
class A { /* ... */ };
long compare (A const&, A const&);
class B : public A { /* ... */ };
ISortedSet < B > BSet;
```

The instantiated default compare() function for class B uses the operator< of B, if defined. Otherwise, a compilation error occurs, because class B's operator< is not found. You must define standard functions such as equal() or compare() for the actual element type B to prevent the template instantiation of those functions, in case you want to provide a class-specific equal() or compare() function for B.

The classes IElemPointer, IMngElemPointer, and IAutoElemPointer (see “Managed Pointers” on page 119) internally use a function called elementForOps() to direct functions such as equal() and compare() to the referenced element, so that they are not applied to the pointer itself and so that instantiations such as ISet <IElemPointer

<Task>> perform the functions on the elements. This indirection is usually transparent but you must consider it when you derive classes from the IElemPointer class. The standard operation classes first apply a function elementForOps() to the element before they apply the corresponding non-member (equal(), ...) function. By default, a corresponding template function is instantiated for elementForOps() which takes an element as input and returns that element. For pointer classes that perform operations on the pointers themselves (IAutoPointer, IMngPointer), this function takes the pointer as input and returns the same pointer. For pointer classes that perform the operations on the referenced elements (IElemPointer, IAutoElemPointer, IMngElemPointer), this function takes the pointer as input and returns the referenced element. If a class derived from IElemPointer<E> is used as a collection element type, the default template functions must be instantiated before a conversion will be considered. A derived class must therefore explicitly redefine the elementForOps() function, as shown in the following example, where class TaskPtr redefines both versions of elementForOps() by calling the default elementForOps() with a TaskPtr as argument. Both versions are then made to return a cast to Task reference:

```
class TaskPtr : public IElemPointer < Task >  {
   friend Task& elementForOps ( TaskPtr & t ) {
      return ( Task & ) elementForOps ( t ); }
   friend Task const & elementForOps ( TaskPtr const & t ) {
      return ( Task const & ) elementForOps ( t ); }
};
ISet < TaskPtr > taskSet;
```

Using Pointer Classes

In C++, variables and function arguments have their values copied when they are assigned. This copying can decrease a program's efficiency, especially when the objects are large. To improve efficiency, pointers or references are often used for common objects. For example, a pointer or reference to the object can be copied, instead of the object itself. Polymorphism is achieved with pointers through the use of virtual functions. Pointers to elements can be used as collection element types, rather than the elements themselves. (References are not allowed as collection element types).

The Collection Classes define five pointer classes:

- IElemPointer
- IAutoPointer
- IAutoElemPointer
- IMngPointer
- IMngElemPointer

These types are referred to as *pointer classes*. Their main characteristics are:

Using Pointer Classes

- Certain pointer classes perform storage management. Storage management in this context means that referenced objects are automatically deleted under certain conditions.
- Certain pointer classes, if stored in a collection, perform all element and key-type functions, for example equality test, on the referenced elements, instead of on the pointers themselves.
- Certain pointer classes combine both of the above features.

You can use pointer classes that perform element and key functions on the referenced elements, by storing pointers from these classes in collections. For pointers from pointer classes that perform storage management only, you can use the pointers instead of native C++ pointers for general purposes.

You can store pointers from these pointer classes, as well as C++ pointers, as elements in any collection. The following sections describe the enhancements that pointers from the above classes provide over native C++ pointers.

Overview of Pointer Classes

If you store standard C++ pointers in a collection, the collection performs all element functions (for example, equality test) on the pointers themselves. This is not always what you intend. If you want the collections to perform those element functions on the referenced elements instead, use one of the following pointer classes:

- `IElemPointer`
- `IAutoElemPointer`
- `IMngElemPointer`

If you use pointers from these classes, and you check, for example, the equality of two pointers from your collection of pointers, `True` is only returned if the referenced elements are equal as defined by the equality relation of the element type, even if the elements are located at different addresses in memory. The same equality test for a collection of C++ pointers only returns `True` if the pointers point to the same address.

Pointers from the three `I...Elem...` classes are also called *element pointers*. Element pointers are only useful when you store them in a collection. The elements themselves are not "stored" in the collection, although information from the elements is used by Collection Classes functions. See "Element Pointers" on page 118 for more information on the element pointer types.

Using Pointer Classes

If you prefer to perform all element functions (for example, equality test) on the pointers themselves, and not on the referenced objects (elements), you can use one of the following pointer classes:

- `IMngPointer`
- `IAutoPointer`

For example, if you check the equality of two such pointers from your collection of pointers, `True` is only returned if the pointers point to the same address. (This is the same behavior as you would expect for native C++ pointers.)

Most pointer classes perform automatic storage deallocation for objects that are no longer referenced. They are:

- `IAutoPointer`
- `IAutoElemPointer`
- `IMngPointer`
- `IMngElemPointer`

Pointers of classes `IAuto...` are called *automatic pointers*. They perform memory management so that referenced objects are deleted as soon as the pointer passes out of scope. See "Automatic Pointers" on page 119 for more information on automatic pointers.

Pointers of classes `IMng...` are called *managed pointers*. They perform memory management so that the references to objects are counted, and objects are deleted only when they are no longer referenced by any managed pointer. See "Managed Pointers" on page 119 for more information on managed pointers.

To exploit the advantage of memory management, you can use non-element pointers (for example, `IMngPointer`) instead of standard C++ pointers without storing the pointers in a collection.

Automatic storage management is particularly useful when functions return pointers or references to objects that they have created (dynamically allocated), and the last user of the object is responsible for cleaning up.

The following features of Collection Classes pointer types give you the choices shown in the table below. Standard C++ pointers are included for comparison.

- Element functions performed on referenced elements
- Element functions performed on pointers
- Automatic storage management

Using Pointer Classes

	Destruction of Pointed Objects		
	Not managed	**When out-of-scope**	**Reference counted**
Collections call element operations on pointer	Standard C++ pointer	`IAutoPointer`	`IMngPointer`
Collections call element operations on referenced object	`IElemPointer`	`IAutoElemPointer`	`IMngElemPointer`

The pointer classes can only take arguments of type `class` or `struct`. The reason is that the overloaded `operator->` needs to return an object of such a type. You can apply pointer objects from these five classes in the same way you use ordinary C++ pointers, with the `*` and `->` operators. Elements are implicitly deleted, except in the case of `IElemPointer`. To delete an element referred to by an `IElemPointer`, you must use an explicit conversion to the referenced element type:

```
IElemPointer < E > ptr;
// ...
delete (E*) ptr;
```

Element Pointers

If you create a collection of C++ pointers or pointers of type `IMngPointer` or `IAutoPointer`, Collection Classes methods that use element comparison functions will do the comparison on the pointers to the elements, instead of on the elements themselves.

If you do want element functions to work on the pointers instead of the referenced elements, you do not need to implement equality and ordering relation for the chosen pointer type (`IAutoPointer`, `IMngPointer` or C++ pointers). The compiler can instantiate the default element function templates. If necessary, you can implement your element functions for the referenced element type.

In the following example, adding, locating, and other functions are based on pointer equality and ordering, and not on the equality defined for the `Task` type.

```
class Task
{
    TaskId ivId;
    //...
    IBoolean operator== (Task const& t)
    { return ivId == t.ivId; }
};
typedef IMngPointer < Task > MngPointerToTask;
ISet < MngPointerToTask > setOfTaskPointers;
                           // equality will refer to pointer
                           // though it is defined for Task
```

On the other hand, if you want element functions to work on the elements referenced by the pointers, the Collection Classes offer the IElemPointer, IAutoElemPointer, and IMngElemPointer pointer classes, which are instantiated with the element type. Pointers of these classes automatically apply all element functions, except for assignment, to the referenced object. Element pointers are constructed from C++ pointers. The C++ dereferencing operators * and -> are defined, for element pointers, to refer to the referenced objects.

```
class Task
{
  TaskId ivId;
  Priority priority () const;
};

typedef IElemPointer < Task > TaskPtr;
ISet < TaskPtr > taskSet;       // taskSet is a set of task pointers
TaskPtr t1 (new Task);          // convert a new C++ task pointer to
                                // Collection Class task pointer
taskSet.add (t1);               // add the pointer to taskSet
//...
taskSet.elementAt (cursor)->priority(); // apply priority function
                                // to the element (priority is
                                // a member function of Task)
// ...
taskSet.remove(t1);             // remove pointer from collection
delete (Task*)t1;               // delete task pointed to by t1.
```

The dynamically created elements are not automatically deleted when they are removed from the collection.

Managed Pointers

Managed pointers keep a reference count for each referenced object (element). When the last managed pointer to the object is destructed, the object is automatically deleted. Use managed pointers when you are unsure who is responsible for deleting an object. If several pointers to an object may be introduced over time, the order in which the pointers are released is not known.

The following example shows how to use pointers from the IMngElemPointer class:

```
typedef IMngElemPointer < Task > TaskPtr;
ISet < TaskPtr > tasks;
TaskPtr t1 (new Task, IINIT);
tasks.add (t1);
// ...
tasks.remove (t1);
```

In the example, the allocated task will automatically be deleted by the remove() function unless it is referenced through another TaskPtr.

Automatic Pointers

Automatic pointers do not keep a reference count. A referenced object (element) is automatically deleted in two cases:

- The automatic pointer is destructed. Use automatic pointers when the lifetime of the element is the same as the lifetime of the pointer, and an explicit deletion of the element is awkward or even impossible. This case applies in particular to pointers to objects that are dynamically created within a function, and whose lifetime is the scope of the function. The function may be left through several return statements or through an exception being thrown from some other function being called.
- Using the assignment operator, the automatic pointer is used to point to another element (which is implicitly a new element). The assigned pointer is set to `NULL`.

If you define a collection taking automatic pointers as elements, the elements are automatically deleted when the collection is destructed, when an element is removed, or, if the element was not added to the collection, when the variable or temporary holding the pointer is destructed:

```
{
ISet < IAutoElemPointer < IString > > eSet;
eSet.add (IAutoElemPointer (new IString ("abc"), IINIT));
eSet.add (IAutoElemPointer (new IString ("def"), IINIT));
// the temporary automatic pointer variables have been set
// to NULL when the pointer was copied to the collection

   {
    eSet.add (IAutoElemPointer (new IString ("def"), IINIT));
   } // deletes the second IString ("def"), because it was not added
    // (note that in a set each element occurrs only once)

}  // deletes the IString ("abc") and first IString ("def")
    // with the destruction of the eSet
```

Transfer of Automatic Pointers

You should be aware of the implementation details described below when transferring automatic pointers between functions. Consider the following cases:

- A calling function passes an automatic pointer to a called function and the pointer is returned.

```
IAutoPointer <Int> f (IAutoPointer <Int> i) { return i; }
// ...
main () {
    IAutoPointer <Int> i (new Int (5), IINIT);
    cout << *f(i) << endl;
}
```

  This program results in the following taking place at run time:

  - **`main`** constructs an `IAutoPointer` object `i` and initializes it with the address of `Int` object 5.
  - On invocation of `f()`, the copy constructor of `IAutoPointer` is called, and the new constructed auto pointer is initialized with the address of the given input pointer. The given pointer is set to NULL. On return from `f()`, the copy constructor of `IAutoPointer` constructs a new auto pointer in `main()` and initializes it with the address of the auto pointer object from `f()`, which is then destructed.

- When **main** exits, it calls the destructors for all auto pointer objects and the destructor for Int object 5.

- A called function has no input, but returns an object that has been dynamically been created using an automatic pointer.

```
Int g() {
   IAutoPointer <Int> j (new Int (6), IINIT);
   return *j;
}
// ...
main () {
   cout << g() << endl;
}
```

This program results in the following taking place at run time:

- On invocation of g(), this function constructs an IAutoPointer object, constructs an Int(6) object, and initializes the auto pointer with the address of Int(6).
- On return from g(), the copy constructor of Int constructs a new Int(6) object in main(). The auto pointer object and the Int(6) object in g() are destructed.
- On exit from main(), the Int(6) object is destructed.

Constructing Pointers from the Pointer Classes

All pointers from the pointer classes have two constructors: a default constructor that initializes the pointer to NULL, and a constructor taking a C++ pointer to an element that you must have created before (using new).

Implicit conversions from a C++ pointer to a managed or automatic pointer are dangerous: elements might be implicitly deleted without your being aware that they have been. Therefore, the conversion functions for these classes take an extra argument IINIT to make the construction explicit. Hence, the notation for creating a managed or automatic pointer is:

```
IAutoPointer < E > ePtr (new E, IINIT);
```

Note: After you have constructed a managed or automatic pointer from a C++ pointer, do not use the C++ pointer. You should only access the element through the pointer of the given class. Otherwise, the element could be implicitly destructed while a C++ pointer still refers to it. In particular, you must not construct two managed pointers or two automatic pointers from the same C++ pointer, because the managed pointers would keep two separate reference counts, and to implicitly delete the referenced element twice. For example:

```
IString *s = new IString("...");
IMngPointer < IString > p1 (s, IINIT); // OK
IMngPointer < IString > p2 (s, IINIT); // NO!
// Do not use s a second time, because the compiler may try to
// delete the IString object referred to by s, p1, and p2 twice.
```

Using Pointer Classes

> You should keep the following rule in mind when using managed or automatic pointers created from standard pointers: *Never use the C++ pointer once the managed or automatic pointer has been created from it,* because this may interfere with the automatic storage management. For example, the object that is referenced by a C++ pointer and by an automatic pointer created from this C++ pointer is deleted as soon as the automatic pointer gets out of scope. The C++ pointer then points to undefined storage.

The extra IINIT argument is introduced to make such situations explicit and especially to avoid the usage of the constructor as an implicit conversion operator. The IINIT argument is defined as follows:

```
enum IExplicitInit {IINIT};
```

Without the IINIT argument, you might try to write code such as the following:

```
typedef  IMngPointer < Task >  TaskPtr;
void   func ( TaskPtr currentTask );
// ...
Task* stndP =  new Task;           //creating a C++ pointer
TaskPtr mngdP = TaskPtr ( stndP ); // creating a managed pointer from
                                   // a C++ pointer
func ( stndP );   // Error: Second use of the C++ pointer
```

For the call to func(), the compiler would call a constructor for implicit conversion if the constructor did not require IINIT. On function return, the temporary managed pointer would be destructed and the Task object deleted.

Notes on Pointer Classes

1. The pointer classes do not work with basic types such as **int**, **long**, and **char**.
2. If you implement a key collection containing element pointers, you must define your key() function with the element as input, not the pointer to the element, for example:

   ```
   typedef IKeySortedSet <IMngElemPointer <Element>, int> keySortedSetOfPointers;
   // ...
   int const& key(Element const& element) {
      return element.elementKey();
     }
   ```

   where elementKey() returns the element's key.
3. The copy constructor and assignment operator of an automatic pointer are defined in a way that resets the source pointer to NULL. This prevents multiple automatic pointers from pointing to the same element. In the following example, p2 is implicitly set to NULL:

   ```
   IAutoPointer < E > p1, p2;
   ...
   p1 = p2;
   ```

   However, the copy constructor and assignment operator still take a const argument (using a const cast-away) to maintain compliance with the standard interface for these operations. This standard interface is required, for example, when you use these types as element types in collections, because the copy constructor and assignment operator are required to have such an interface. (If this interface were not a requirement, the collection's `add()` function could not take a const argument.)

Using Pointer Classes

Tailoring a Collection Implementation

This chapter describes how to tailor a collection implementation for your specific applications. It describes the based-on concept and predefined implementation variants.

Introduction

When you are developing a program that uses a collection, you should begin by using the default implementation and go on to a final tuning phase where you choose implementations according to the actual requirements of your application. You can determine these requirements by profiling or by using other measurement tools. This section describes how to choose between a variety of implementations provided by the Collection Classes as well as how to create your own implementation classes.

As described in "The Overall Implementation Structure" on page 88, each abstract data type has several possible implementations. Some of these implementations are *basic*; that is, the collection class is implemented directly as a concrete class. These basic implementations include:

- AVL trees
- Hash tables
- Linked sequences
- Tabular sequences

Other implementations, including bags, are *based on* other collection classes. The based-on concept provides a systematic framework for choosing the most appropriate implementations. It is also useful for extending the Collection Classes with other basic implementations, such as specific kinds of search trees, and for using these implementations as the basis for other data abstractions such as sets, maps, and bags.

Replacing the Default Implementation

You can easily replace the default implementation with another implementation. Suppose that you have a key set class called MyType that has been defined with the default implementation IKeySet. The definition of this class would look like this:

```
typedef IKeySet < Element, Key > MyType;
```

If you want to replace the default implementation, which uses an AVL tree, with a hash table implementation, you can replace the above implementation with the following definition:

```
   typedef IHashKeySet < Element, Key > MyType;
```

If you replace a collection's default implementation with one of its implementation variants, you must determine what element functions and key-type functions need to be provided for the variant. You must then provide those functions. The list of required functions is not always the same for a collection's default implementation as for particular implementation variants. Required functions for a collection's default implementation or an implementation variant are listed in the collection's chapter in the *Open Class Library Reference*. See the section “Template Arguments and Required Functions” in each such chapter.

The Based-On Concept

The Collection Class Library achieves a high degree of implementation flexibility by basing several collection class implementations on other abstract classes, rather than by implementing them directly through a concrete implementation variant of the class. This design feature results in an implementation path rather than the selection of an implementation in a single step. The Collection Class Library contains type definitions for the most common implementation paths; they are described in the corresponding sections of the *Open Class Library Reference*. See Figure 12 on page 129 for an illustration of implementation paths. The figure is explained in “Provided Implementation Variants” on page 128.

The element functions that are needed by a particular implementation depend on all collection class templates that participate in the implementation. While `ISet` requires at least element equality to be defined, an AVL tree implementation of this set also requires the element type to provide a comparison function. A hash table implementation also requires the element type to have a hash function. The required element functions for all predefined implementation variants are listed in the chapters for individual collection types in the *Open Class Library Reference*.

For a concrete implementation, such as a set based on a key-sorted set that is in turn based on a tabular sequence, these class templates have to be *plugged* together. The plug mechanism requires class templates to be used as template arguments. Because C++ does not allow class templates as template arguments, the Collection Class Library implements the plug mechanism using macros. Two macros are provided:

- One for defining a template with an additional operations class argument, for example, `IDefineGSetOnGKeySortedSet`. See “Using Element Operation Classes” on page 110 for information on why you would use such additional arguments.
- One for defining a template with only the element type, or the element and key types, as arguments, for example, `IDefineCollectionWithOps` and `IDefineKeyCollectionWithOps`.

The second macro needs, as an element operations class, an argument as is described in “Using Element Operation Classes” on page 110. Standard operations class templates are predefined that can be used for this purpose. Their names are systematically derived from the operations they define. The name structure is:

```
I<elem-ops>[K<key-ops>]Ops
```

where <elem-ops> and <key-ops> are a sequence of letters: E for equality, C for comparison, and H for hashing. IEKEHOps, for example, is an operations template that provides element equality, key equality, and hashing on keys as well as the basic memory management and element assignment operations.

For the following example, assume you have a specific form of a sorted tree called IGMySortedTree, which is a new implementation for a KeySorted Set. It must exactly implement the interface provided for a KeySortedSet:

- It must have three template arguments, the element type, the key type, and an element operations class.
- It must implement all of the member functions defined for KeySortedSet.

A set implementation IGMySet that is based on this new sorted tree is defined as follows:

```
IDefineGSetOnGKeySortedSet (IGMySortedTree, IGMySet)
```

IGMySet is defined as a template with the element type and an element operations class as arguments. A template that only takes the element type as argument and that uses the standard element operations for equality and comparison can then be defined:

```
IDefineCollectionWithOps (IGMySet, IECOps, IMySet)
```

This expands to the code shown below. The expansion is not intended to give you an in-depth understanding of how the mechanism works internally. It merely illustrates the value of the macros.

```
template < class Element, class ElementOps >
class IGMySet :
IWSetOnKeySortedSet
   < Element, ElementOps,
     IGMySortedTree
       < Element, Element,
         IOpsWithKey < Element, ElementOps > > >
{ /* ... constructor redefinition ... */ };
template < class Element >
class IMySet : public IGMySet < Element, IECOps < Element > >
{ /* ... constructor redefinition ... */ };
```

Provided Implementation Variants

Figure 12 on page 129 lists the basic and based-on implementations provided by the Collection Classes. The upper left corner of each cell contains the name of the (abstract) collection class; basic implementations are written in smaller letters in bold face, while based-on implementations are described by arrows starting from the class that they implement and ending in the (abstract) class on which they are based. An implementation choice for a given class must use either a basic implementation for this class or follow a based-on implementation path that ultimately leads to a basic implementation.

Take the example of the `Bag` abstraction. The `Bag` is not implemented directly. (You can tell this from the figure because no implementation variant name appears in bold in the box containing `Bag`.) To determine the possible implementation variants for `Bag`, follow the arrows out of the `Bag` box:

- One arrow leads to the KeySet box. The KeySet box contains an implementation variant, **HashTable KeySet**, so this is one possibility. An arrow also points from the KeySet Box to the KeySortedSet box, which allows the following possibilities:
  - **AVL Tree Key Sorted Set** (appears in KeySorted Set box)
  - **B* Tree Key Sorted Set** (appears in KeySorted Set box)
  - An arrow leads from KeySorted Set to Sequence, which contains the following implementation variants:
    - **LinkedSequence**
    - **TabularSequence**
    - **DilutedSequence**

A `Bag` can therefore be implemented using any of the six implementation variants cited in bold face above.

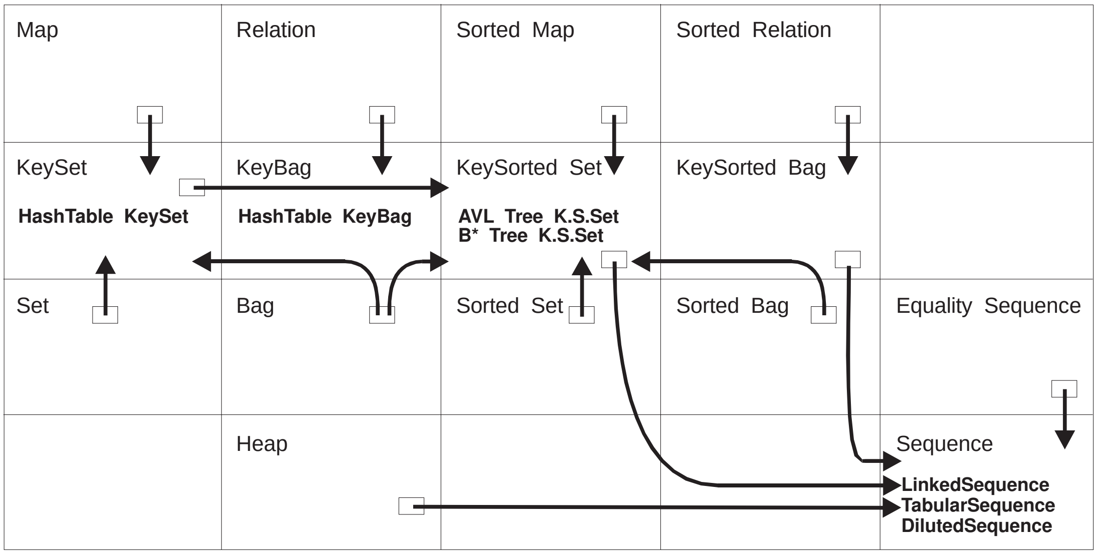


Figure 12. Possible Implementation Paths

The following table lists the based-on implementations of the Collection Classes, and the header files that provide the `IDefine...` macros. Usually, you do not need to use the `IDefine...` macros. You can use them, however, to define your own implementation variant for a collection class and to integrate it into the scheme of implementation paths shown in Figure 12.

Macro Name	Header File
`IDefineGBagOnGKeySet`	`ibagks.h`
`IDefineGBagOnGKeySortedSet`	`ibagkss.h`
`IDefineGDequeOnSequence`	`ideqseq.h`
`IDefineGEqualitySequenceOnGSequence`	`iesseq.h`
`IDefineGHeapOnGSequence`	`iheapseq.h`
`IDefineGKeySetOnGKeySortedSet`	`ikskss.h`
`IDefineGKeySortedBagOnGSequence`	`iksbseq.h`
`IDefineGKeySortedSetOnGSequence`	`ikssseq.h`
`IDefineGMapOnGKeySet`	`imapks.h`
`IDefineGMapOnGKeySortedSet`	`imapkss.h`
`IDefineGPriorityQueueOnGKeySortedBag`	`ipquksb.h`
`IDefineGQueueOnSequence`	`iqueseq.h`
`IDefineGRelationOnGKeyBag`	`irelkb.h`

Provided Implementation Variants

Macro Name	Header File
`IDefineGSetOnGKeySet`	`isetks.h`
`IDefineGSetOnGKeySortedSet`	`isetkss.h`
`IDefineGSortedBagOnGKeySortedSet`	`isbkss.h`
`IDefineGSortedMapOnGKeySortedSet`	`ismkss.h`
`IDefineGSortedRelationOnGKeySortedBag`	`isrksb.h`
`IDefineGSortedSetOnGKeySortedSet`	`isskss.h`
`IDefineGStackOnSequence`	`istkseq.h`

Features of Provided Implementation Variants

You can implement a given collection type (bag, key sorted set, etc.) in a number of different ways. 🕮 The possible implementation variants are described in "Provided Implementation Variants" on page 128, and are listed in the "Class Implementation Variants" section of each collection chapter in the *Open Class Library Reference*. The Collection Classes provide multiple implementation variants for collections because different variants have different performance and storage use characteristics. After you have coded and debugged an application that uses the Collection Classes, you can change an implementation to a variant that is well-suited to the ways in which you use the collection. For example, in "Key Set" in the *Open Class Library Reference*, the section "Variants and Header Files" on page 159 lists six implementation variants, including the default key set. These variants are implemented using the following concrete techniques:

- AVL tree (the technique used for the default key set)
- B* tree
- Hash table
- Sorted linked sequence
- Sorted tabular sequence
- Sorted diluted sequence

As it turns out, the implementation variants for key set encompass all the concrete techniques used by the Collection Classes. Other collections may only use some of the techniques in the list above. If you want to choose the best implementation variant for your program, you need to know the advantages of each concrete technique. The remainder of this section describes each technique and presents its advantages and the trade-offs it entails.

Sequences

Sequences are generally used to store elements sequentially. Each of the three available implementation variants for sequences allows certain operations to be done more efficiently than others. The benefits of each variant are described first, and then each variant is explained in detail.

> *Tabular sequences* provide good performance where a collection is primarily used for reading data but elements are not frequently added or deleted once the collection is created.
>
> *Diluted sequences* are more suitable for collections where some elements are inserted or deleted after the collection is created, but where the collection is still primarily read from rather than written to.
>
> *Linked sequences* are more suitable than tabular or diluted sequences when you anticipate that many elements will be added or deleted, and where you cannot accurately predict the maximum size of the collection when it is first created.

Following are descriptions of each type of sequence.

Tabular Sequence

A *tabular sequence* is an array implementation of a list. The elements are stored in contiguous cells of an array. In this representation, a list can easily be traversed, and new elements can easily be added to the tail of the list. If an element needs to be inserted into the middle of the list, however, all following elements need to be shifted to make room for the new element. Similarly, if an element needs to be removed from the list, and the element is not the last element in the list, all elements following the element to be deleted must be shifted in to close up the gap.

A tabular sequence can access all elements quickly because all elements can be stored in a single storage block. If all of the following conditions hold true for your use of a collection, a tabular sequence is a suitable implementation variant to use:

- The elements to be stored are small.
- You can predict with some accuracy how many elements your application will have to handle.
- Few or no elements will need to be added or deleted once the collection is first created.

Note that memory is statically allocated for tabular sequences, at the beginning of your program.

Figure 13 on page 132 shows a tabular sequence implementation variant.

Provided Implementation Variants

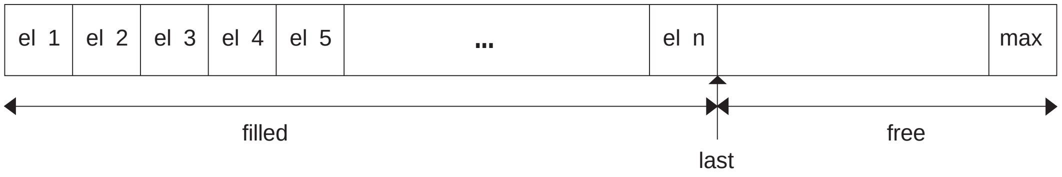


Figure 13. Tabular Sequence Implementation Variant

Diluted Sequence

A *diluted sequence*, like a tabular sequence, is an array implementation of a list. However, when you delete an element from a diluted sequence, it is not actually deleted, but only flagged as deleted. This provides a performance advantage, in that elements following a deleted element do not need to be shifted. The additional overhead of using a dilution flag is trivial.

If you want to add a new element at a certain position, only those elements between that position and the next element flagged as deleted need to be shifted. (If no elements later in the list are flagged as deleted, then all elements beyond the insertion position must be shifted.)

Use a diluted sequence rather than a tabular sequence if your application will be doing much adding or deleting of elements after the collection is established.

Figure 14 shows a diluted sequence implementation variant.

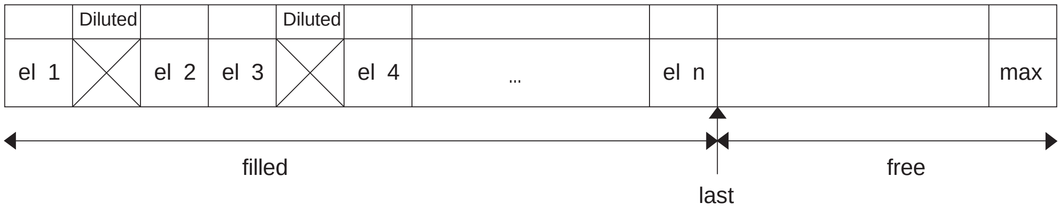


Figure 14. Diluted Sequence Implementation Variant

Linked Sequence

A *linked sequence* uses pointers to link each element to its predecessor and successor. This implementation does not require contiguous memory for storing an array, which means that elements do not have to be shifted to make room for new elements or to close up gaps created by deleted elements.

Because storage is dynamically allocated and freed, this implementation variant is a good choice in applications that add or delete many elements, particularly where you cannot predict the amount of storage required. Figure 15 on page 133 shows a linked sequence implementation variant.

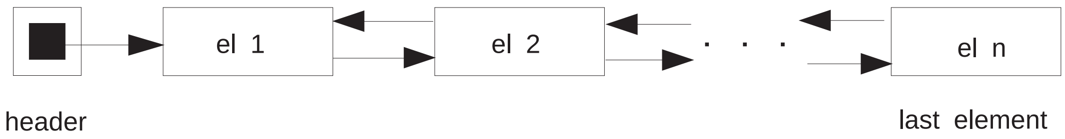


Figure 15. Linked Sequence Implementation Variant

Trees

A tree is a collection of nodes. The nodes either contain the data of the collection or pointers to that data.

A node normally contains a reference to one or more other nodes. Referenced nodes are *children* of the referencing node. One node is the entry point to the tree. This node is designated as the *root*. Nodes without any references to other nodes are called *leaf nodes* or *terminal nodes*.

Trees in general are more useful for searching elements than for adding and deleting elements. For this reason, they are often called *search trees*. The descriptions of AVL and B* trees below explain why trees are well-suited for searching.

AVL Tree

AVL trees are a special form of binary tree. You can better understand AVL trees if you know how a binary tree is structured.

Trees are *binary trees* when all nodes have either zero, one, or two children. Binary trees are often used in applications where you want to store elements in a certain order. In such cases, the left child always points to an element that comes earlier in the order than the parent node, and the right child points to an element that comes later than the parent. A search through a binary tree begins at the root node. The search then continues downward until the desired element is found, by determining whether a node comes before or after the searched-for node, and then following the appropriate branch. For example, the binary tree shown in Figure 16 on page 134 has elements added in the following sequence: *8 - 10 - 5 - 1 - 9 - 6 - 11*. A search for element 9 begins at the root node (element 8). Assuming that the element value defines the ordering relation, the search would take the right node from element 8 (because 9 is greater than 8) and would arrive at element 10. The search would take the left node from element 10 (because 9 is smaller than 10) and would arrive at element 9, the desired element.

Provided Implementation Variants

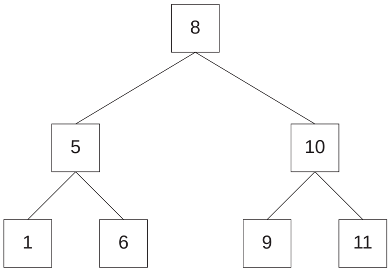


Figure 16. Binary Search Tree

One drawback of a binary search tree is that the tree can easily become unbalanced. Figure 17 shows how unbalanced the tree becomes when the elements 12 through 15 are added.

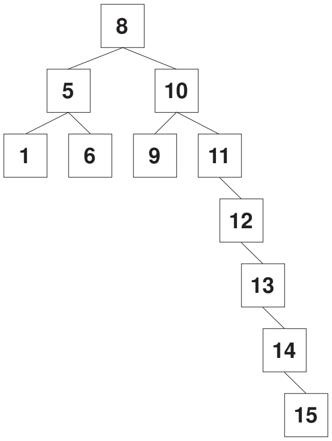


Figure 17. Unbalanced Binary Search Tree

This tree looks almost like a linked sequence, without the performance advantage of a normal binary search tree. To obtain this performance advantage, a binary search tree should always remain balanced. The *AVL Tree* is a special form of binary search tree that maintains balance.

The *AVL tree* was invented by the two mathematicians, Adel'son-Vel'skii and Landis, from whom it derives its name. AVL trees are *height-balanced.* They have the property that, for every node in the tree, the height of that node's left subtree minus the height of the right subtree is always -1, 0, or +1. AVL trees provide better performance than ordinary binary search trees because they do not become unbalanced. Unbalanced trees often have very poor search characteristics. If adding or removing an element from an AVL tree causes the tree to lose its AVL property, then a few local readjustments are sufficient to restore the AVL property. Figure 18 on page 135 shows how the unbalanced tree shown earlier would look after the AVL property is restored.

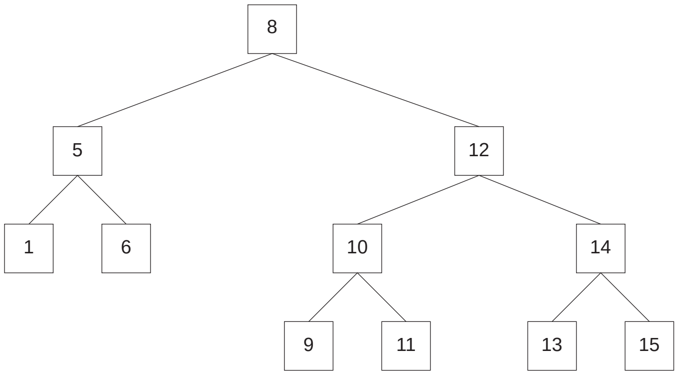


Figure 18. AVL Tree

AVL trees are useful for collections containing a large number of small elements. An AVL tree implementation is even suitable for adding and deleting, because the performance overhead for the rebalancing that sometimes occurs when an element is added or deleted is still less expensive than searching through the elements of a sequence to find the position at which to add or delete an element.

If you use a set collection and do not choose an implementation variant, you are automatically using an AVL tree. If you use a set and are not aware that the set is implemented as an AVL tree, you may be surprised that a set requires an ordering relation, when a set is an *unordered* collection, as shown in Figure 7 on page 81. The reason a set requires an ordering relation is that an AVL tree requires an ordering relation so that it knows where to add new elements or where to find elements being accessed or deleted. As this example shows, required element and key-type functions are determined by two factors:

- Some functions are required because of the properties of the collection.
- Some properties are required because of the implementation variant you choose.

B* Tree

A *B* tree* is a search tree that may have more than two references per node. Figure 19 on page 136 shows a B* tree with up to five children per node.

Provided Implementation Variants

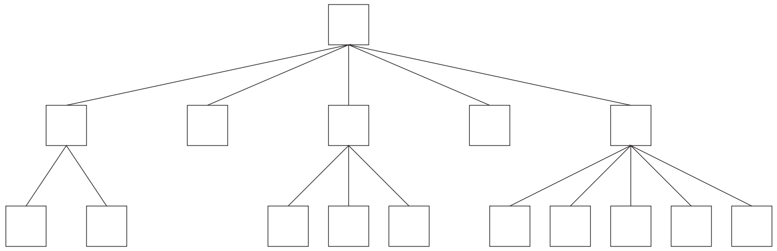


Figure 19. A B tree*

A B* tree combines the advantages of binary search and sequential access upon the same set of keys. B* trees are based on two simple ideas:

- The internal nodes are used only for storing the keys, with all real data stored at the leaves. A B* tree takes into consideration the page or block size of the operating system's virtual memory structure, and is suitable for applications where paging or memory thrashing is a constraint.
- The leaves of a B* tree are chained together in logical sequence to support sequential access.

A B* tree implementation variant is suitable when you have many large elements that are accessed by key. Because keys and their data are separated, the keys in the tree structure are used for a quick search and the pointers are used for quick access to the data.

In contrast to a B* tree, keys and data in an AVL tree are both stored in the nodes. This means that searching through elements could cause page faults if the elements are large, because the various keys may be spread across several pages along with the data they refer to.

In Figure 20 on page 137, the B* tree has an order of 5 (which means that each internal node has a maximum of five references). The data is stored only in the leaves. A leaf block is built to hold one element. A leaf block may be larger than one page. The B* tree implementation uses the keys in the nodes for quick access to a required page (leaf), or it uses the keys for a quick sequential access to all pages, and hence to all elements.

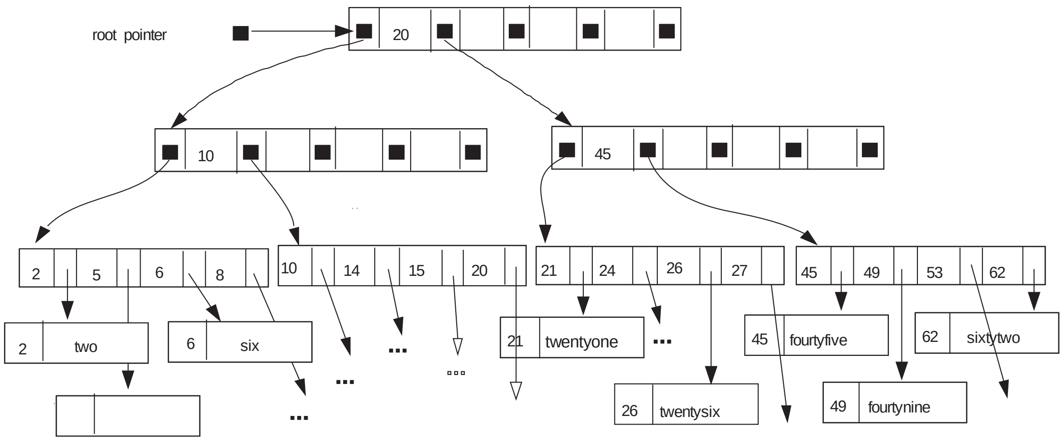


Figure 20. B Tree Implementation Variant*

Hash Table

Hashing is another important and widely used technique to implement collections. Conceptually, hashing involves calculating an index from the key or other parts of an element, and then using that index to look for matches in a hash table. The function that calculates the index is called a *hash function*.

A hash table implementation variant is suitable for nearly all applications with a balanced mix of operations. Such an implementation is quick for retrieving elements. It can also add and delete elements quickly, because, unlike an AVL tree, it does not need to be rebalanced. The efficiency of a hash-table implementation is largely dependent on how efficiently you implement the hash function.

You cannot use a hash-table implementation variant when you require your elements to appear in main storage in sorted order (where elements earlier in the sorting order have lower addresses than elements later in the sorting order). On the other hand, you must use a hash table if you have a complex key (one composed, for example, of several attributes of an element), and either you cannot find a reasonable way to compare keys, or the comparison would be expensive.

For collections that do not provide access by key, but that support a hash-table implementation variant, the complete element is used as the input to the hash function.

Provided Implementation Variants

Hashing, as implemented in the collection classes, allows elements to be stored in a potentially unlimited space, and therefore imposes no limit on the size of the collection. Figure 21 on page 138 shows a hash table implementation variant.

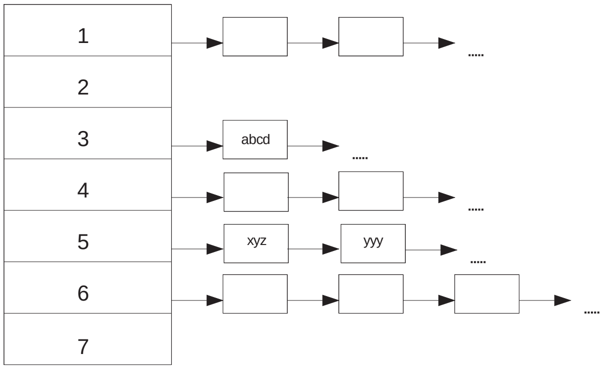


Figure 21. Hash Table Implementation Variant

The hash function that calculates the index *3* from *abcd* is implemented as follows:

1. Each character is transformed into an integer according to its position in the alphabet.
2. The resulting integers are added together.
3. The result is divided by the hash table size. The remainder is the hash.

This hash function returns the following results for elements *abcd*, *xyz* and *yyy*:

- *abcd*: (1 + 2 + 3 + 4) % 7 = 3
- *xyz*: (24 + 25 + 26) % 7 = 5
- *yyy*: (25 + 25 + 25) % 7 = 5

The principal behind a hash table is that the possibly infinite set of elements in your collection is partitioned into a finite number of hash values (*1, 2, 3, ...*). Your hash function is called with a key and a modulo value, and you use the key and the modulo value to arrive at an integer hash value. If for two different keys the hash function returns the same hash value (as for *xyz* and *yyy* in the previous figure), a hash *collision* occurs. In such cases, a hash implementation constructs a collision list where all keys returning the same hash value are linked.

In the best case, for each different key, your hash function should return a different hash value. At the very least, it is desirable for the collision lists to remain small so that access time is fast. This means that hash values should be evenly distributed. Your hash function should randomly hash the key so that the hash value is not dependent on the key value in any trivial way. Your hash function should always return the same hash value for a given key and modulo provided to it.

Polymorphic Use of Collections

This chapter describes how you can use polymorphism in the Collection Classes.

Introduction to Polymorphism

Polymorphism allows you to take an abstract view of an object or function argument and use any concrete objects or arguments that are derived from this abstract view. The collection properties defined in "Flat Collections" on page 80 define such abstract views. They are represented in the form of the class hierarchy in Figure 11 on page 92.

Polymorphic use of collections differs from polymorphism of the element type. Element polymorphism means that you can use the collections with any elements that provide basic operations like assignment and equality; this kind of polymorphism is implemented by the use of the C++ template concept. This chapter deals with the polymorphic use of collections rather than elements. Polymorphic use of collections means that a function can specify an abstract collection type for its argument, for example `IACollection`, and then accept any concrete collections given as its actual argument.

Each abstract class is defined by its functions and their behavior. The most abstract view of a collection is a container without any ordering or any specific element or key properties. Elements can be added to a collection, and a collection can be iterated over. A polymorphic function on collections might be to print all elements; such a function is given as an example on page 140.

Collections whose elements define equality or key equality provide, in addition to the common collection functions, functions for retrieving element occurrences by a given element or key value. Ordered collections provide the notion of a well-defined ordering of the element occurrences, either by an element ordering relation or by explicit positioning of elements within a sequence. They define operations for positional element access. Sorted collections provide no further functions, but define a more specific behavior, namely that the elements or their keys are sorted.

These properties are combined through multiple inheritance: the abstract collection class `IEqualitySortedCollection`, for example, combines the abstract concepts of element equality and of being sorted, which implies being ordered. If a polymorphic function uses this class as its argument type, the arguments will be sorted, and the function can use functions like `contains()` that are only defined for collections with element equality.

Using Reference Classes

For performance reasons explained in “The Overall Implementation Structure” on page 88, concrete collection classes are not directly derived from abstract classes. Instead, you must use an indirection that “couples” a concrete collection with an abstract class. For each leaf in the collection class hierarchy, IASet, for example, there is an indirection class template called IRSet. (📖 See Figure 10 on page 90.) It takes as template arguments the element type and, for key collections, the key type and a concrete collection class that has been instantiated with this element and key type. Instances of this indirection class refer to instances of the concrete collection class. (The R in IRSet stands for reference.) The IR... classes are derived from the corresponding abstract IA... classes and are therefore part of the Collection Class hierarchy. Instances of this class can be used wherever an instance (pointer or reference) of an abstract base class is required.

The following example defines a universal printer class that accepts an arbitrary collection of jobs and prints their IDs. The elements are printed in the iteration order that is defined for the given collection. The concrete key set running cannot be used as an argument to the printer directly, because IKeySet is not derived from the abstract collection classes. The reference class IRKeySet is used for this purpose. It is instantiated with the element and key type, and with the concrete collection class JobSet. An instance of this class refRunning is defined by providing running as a constructor argument. refRunning can then be used as argument to the universal printer.

```
class JobPrinter {
public:
     print (IACollection < Job* > const& jobs)
     { cout << "ID      ..."
       ICursor *cursor = jobs.newCursor ();
       cout << "{ ";
       forCursor (*cursor)
         cout << jobs.elementAt (*cursor)->id() << ' ';
       cout << "}\n";
       delete cursor;
     }
};
// ...
typedef IKeySet < Job*, JobId > JobSet;
JobSet running;
// ...
IRKeySet < Job*, JobId, JobSet > refRunning (running);
JobPrinter jobPrinter;
jobPrinter.print (refRunning);
```

12 Support for Visual Builder for C++

The Collection Classes include special classes that support Visual Builder (Visual Builder is described in *Visual Builder User's Guide*). For every concrete flat collection class (for example `ISequence`), there is a corresponding Visual Builder collection class starting with `IV` (for example IVSequence).

All collection methods that modify a collection send Visual Builder notifications to observers. The class `IPartCollectionNotification` defines four notification IDs for Collection Classes:

`IPartCollectionNotification::addedId` Sent if an element is added to the collection

`IPartCollectionNotification::removedId` Sent if an element is removed from the collection

`IPartCollectionNotification::replacedId` Sent if an element is replaced in the collection

`IPartCollectionNotification::modifiedId` Sent if a collection is changed in any way other than those mentioned above.

For notifications `addedId`, `removedId` and `replacedId`, you can use `INotificationEvent::eventData()` to access event data generated by collections. This event data is an object that includes a `cursor()` method to access a collection cursor. The cursor points to the element referred to by the modification method. For example, if `addedId` is the notification, the cursor points to the added element. The `replaceId` notification also gives you access to a copy of the element that was replaced.

Collection notifications `addedId`, `removedId` and `replacedId` pass a pointer to a class corresponding to the notification.

Notification	**Class**
`addedId`	`IPartCollectionAddedEventData`
`removedId`	`IPartCollectionRemovedEventData`
`replacedId`	`IPartCollectionReplacedEventData`

These classes provide the following methods:

Table 4. Methods of IPartCollection...EventData

Class	Methods
IPartCollectionAddedEventData	const ICursor& cursor() const
IPartCollectionRemovedEventData	const ICursor& cursor() const
IPartCollectionReplacedEventData	const ICursor& cursor() const Element& replacedElement() const

For all notifications except RemovedId, the library sends notification after the modification occurs. The library sends RemovedId notification before the collection is changed because otherwise you would not be able to use the cursor to refer to the element being removed.

Notifications are only sent if the collection is changed by the method. The following methods do not create a notification:

- removeAll() for an empty collection
- add(), when add() does not actually add an element (for example because the element already exists in a unique collection, or because the collection is full)
- remove() if the element is not in the collection
- locateOrAdd if the element is already in the collection

Header Files for Visual Builder Support

The classes IPartCollectionAddedEventData, IPartCollectionRemovedEventData and IPartCollectionReplacedEventData are defined in ipartccl.h. The notificationIds are defined in ipartccn.h.

Example for IVSequence<IString>

The following example demonstrates the use of collection event data for a sequence of IStrings. IString is the main string handling class provided by IBM Open Class. See Chapter 17, “String Classes” on page 197 for information on how to use this class.

```
#include <ivseq.h>
#include <ipartccn.h>
#include <ipartccl.h>

   IObserver &dispatchNotificationEvent(const INotificationEvent&
   anEvent) {

      // Process addedId notification

      if ( anEvent.notificationId() ==
           IPartCollectionNotification::addedId) {
```

```
  cout << "Add at position : "
       << (*(IVSequence<IString>*)&(anEvent.notifier()))
           .position(((IPartCollectionAddedEventData<IString>*)
              ((char*)anEvent.eventData()))->cursor())
       << endl;
  cout << "New Data : "
       << (*(IVSequence<IString>*)&(anEvent.notifier()))
          .elementAt(((IPartCollectionAddedEventData<IString>*)
          ((char*)anEvent.eventData()))->cursor())
       << endl;

// Process replacedId notification

if ( anEvent.notificationId() ==
     IPartCollectionNotification::replacedId) {
  cout << "Replace at position : "
       << (*(IVSequence<IString>*)&(anEvent.notifier()))
          .position(((IPartCollectionReplacedEventData<IString>*)
          ((char*)anEvent.eventData()))->cursor())
       << endl;
  cout << "New Data : "
       << (*(IVSequence<IString>*)&(anEvent.notifier()))
          .elementAt(((IPartCollectionReplacedEventData<IString>*)
          ((char*)anEvent.eventData()))->cursor())
       << endl;
  cout << "Old Data : "
       << ((IPartCollectionReplacedEventData<IString>*)
       << ((char*)anEvent.eventData()))->replacedElement()
       << endl; }
```

13 Exception Handling

This chapter describes the exception-handling facilities provided by member functions of the Collection Class Library. This chapter includes the following topics:

- Introduction to exception handling
- Preconditions and defined behavior
- Levels of exception checking
- List of exceptions
- The hierarchy of exceptions

Introduction to Exception Handling

The C++ exception-handling facilities allow a program to recover from an *exception*. An exception is a user, logic, or system error that is detected by a function that does not itself deal with the error, but passes the error to a handling function. Exceptions can result from two major sources:

- The violation of a precondition
- The occurrence of an internal system failure or system restriction

In this chapter, two kinds of functions are discussed. A *called* function is a Collection Class function that may throw an exception. A *calling* function is a function that calls a Collection Class function. The *calling* function may be a Collection Class function or a function you have defined.

Exceptions Caused by Violated Preconditions

A *precondition* of a called function is a condition that the function requires to be true when it is called. The calling function must assure that this condition holds. The called function implementation may assume that the condition holds without further checking it. If a precondition does not hold, the called function's behavior is undefined.

If you want to make your programs more robust and to locate errors in the test phase, the functions your program calls should check to ensure that their preconditions hold. The Collection Class Library enables this checking through macro definitions. Because this checking often requires significant overhead, it is turned off by default. You need only use it while you are testing the system and verifying that preconditions are always met.

A call to a function that violates the function's preconditions has two possible results:

- If the called function checks its preconditions, the function will throw an exception.
- If the function does not check its preconditions, the behavior of the function is undefined.

Exceptions Caused by System Failures and Restrictions

System failures and restrictions are different from precondition violations. You cannot usually anticipate them, and you have no opportunity to verify that such situations, for example storage overflow, will not occur. These exceptions need to be checked for, and an exception should be thrown if they occur.

Precondition and Defined Behavior

Exceptions are not generally used to change the flow of control of a program under normal circumstances. An example of using exceptions under normal circumstances is a function that iterates through a collection, and exits from the iteration by checking for the exception that is thrown when an invalid cursor is used to access elements. When the iteration is complete, the cursor will no longer be valid, and this exception will be thrown. This is not a good programming practice. A function should explicitly test for the cursor being valid. To make this possible, a function must efficiently test this condition (`isValid()`, for the cursor example).

There are situations where the test for a condition can be done more efficiently in combination with performing the actual function. In such cases, it is appropriate, for performance reasons, to make the situation regular (that is, not exceptional) and return the condition as a IBoolean result. Consider a function that first tests whether an element exists with a given key, and then accesses it if it exits:

```
if (c.containsElementWithKey (key)) {
     // ...
     myElement = c.elementWithKey (key); // inefficient
     // ...
} else {
     // ...
}
```

This solution is inefficient because the element is located twice, once to determine if it is in the collection and once to access it. Consider the following example:

```
try {
    // ...
    myElement = c.elementWithKey (key); // bad: exception expected
    // ...
} catch (INotContainsKeyException) {
    // ...
}
```

This solution is undesirable because an exception is used to change the flow of control of the program. The correct solution is to obtain a cursor together with the containment test, and then to use the cursor for a fast element access:

```
if (c.locateElementWithKey (key, cursor)) {
    // ...
    myElement = c.elementAt (cursor); // most efficient
    // ...
} else {
    //...
}
```

Levels of Exception Checking

Some preconditions are more difficult to check than others. Consider the following possible preconditions:

1. A cursor for a linked collection implementation still points to an element of a given collection.
2. A collection is not empty.

In the production version of a program, it may be less efficient to check the first precondition than the second.

The Collection Class Library provides three levels of precondition checking. They are selected by the following macro variable definitions (use, for example, compile flag `-DINO_CHECKS`):

`INO_CHECKS` Check for memory overflow. Other checks may be eliminated to improve performance.

Default Perform all precondition checks, except the check that a cursor actually points to an element of the collection.

`IALL_CHECKS` Perform all precondition checks, including the (costly) check that a cursor actually points to an element of the collection. This extra check can only fail for undefined cursors.

List of Exceptions

The Collection Class Library defines the following exceptions:

IChildAlreadyExistsException

Occurs when you try to add a child to a tree using `addAsChild()` at a position that already contains a child.

List of Exceptions

ICursorInvalidException

Two cursor properties may lead to the `ICursorInvalidException`:

- Every time a cursor is created, you must specify the collection that it belongs to. If a function takes a cursor as an argument (such as `add()`, `setToFirst()`, and `locate()`), the function can only be applied to the collection that the cursor belongs to. If the function is applied to another collection, the `ICursorInvalidException` results.
- If a function takes a cursor as an input argument (such as `elementAt()`, `removeAt()`, and `replaceAt()`), the cursor must be *valid*. A cursor is valid if it actually refers to some element contained in the collection. You can use the `isValid()` function to determine if a cursor is valid.

IEmptyException

Occurs when a function tries to access an element of an empty collection. Functions that might cause this exception include `firstElement()` and `removeFirstElement()`.

IFullException

Occurs when a function tries to add an element to a bounded collection that is already full. Functions that might cause this exception include `add()` and `addAsFirst()`.

IIdenticalCollectionException

Occurs when the function `addAllFrom()` is called with the source collection being the same as the target collection.

IInvalidReplacementException

Occurs when, during a `replaceAt()` function, the replacing element has different positioning properties (📖 see “Replacing Elements” on page 98) than the positioning properties of the element to be replaced.

IKeyAlreadyExistsException

Occurs when a function attempts to add an element to a map or sorted map that already has a different element with the same key. Functions that might cause this exception include `add` and `addAllFrom()`.

INotBoundedException

Occurs when the function `maxNumberOfElements()` is applied to a collection that is not bounded.

INotContainsKeyException

Occurs when the function `elementWithKey()` is applied to a collection that does not contain an element with the specified key.

IOutOfMemory

Occurs when a function cannot obtain the space that it requires. This exception is not the result of a precondition violation. Functions that add an element to a collection, including `add()` and `addAsFirst()`, can cause this exception.

IPositionInvalidException

Occurs when a function specifies a position that is not valid in a collection. The functions that might cause this exception include `elementAtPosition()`, `removeAtPosition()`, and `setToPosition()`.

IRootAlreadyExistsException

Occurs when the function `addAsRoot()` is called for a tree that already has a root.

The Hierarchy of Exceptions

In the Collection Class Library, all exceptions are derived from the `IException` class described in Chapter 18, "Exception and Trace Classes" on page 213. It provides common functions to access information about an exception that has occurred.

The direct subclasses of `IException` used in the Collection Class Library are `IPreconditionViolation` and `IResourceExhausted`. The following figure shows the hierarchy of exceptions:

Exception Hierarchy

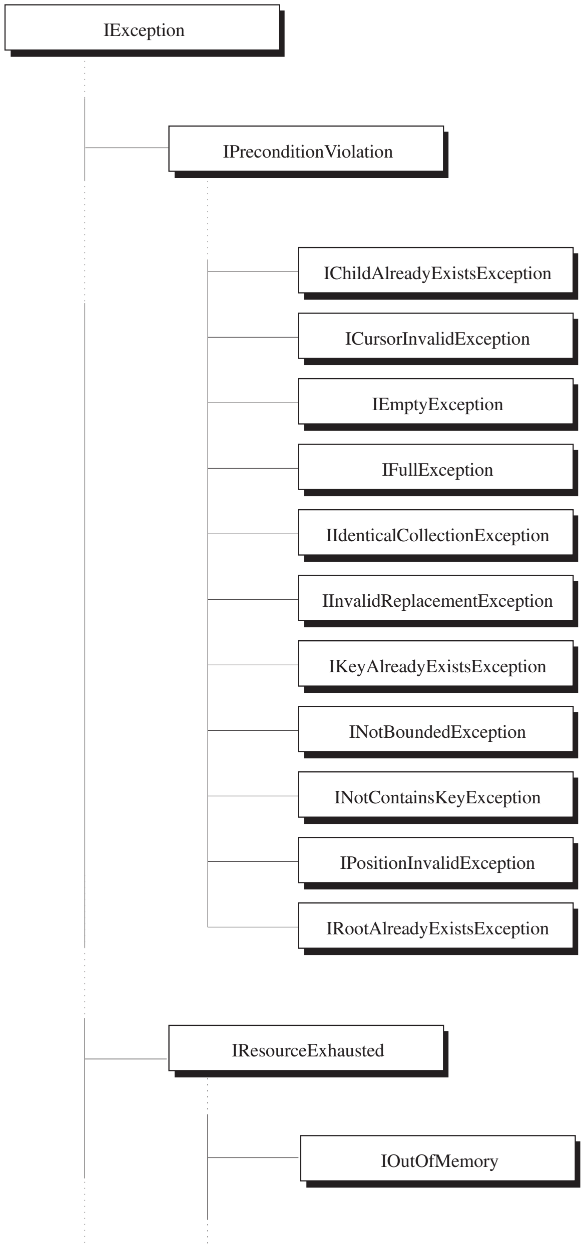

IException
IPreconditionViolation
IChildAlreadyExistsException
ICursorInvalidException
IEmptyException
IFullException
IIdenticalCollectionException
IInvalidReplacementException
IKeyAlreadyExistsException
INotBoundedException
INotContainsKeyException
IPositionInvalidException
IRootAlreadyExistsException
IResourceExhausted
IOutOfMemory

Figure 22. Hierarchy of Exceptions

Collection Class Library Tutorials

This chapter provides a set of tutorial lessons that you can use to learn common Collection Class Library features. Each lesson builds on the lessons you learned and the library features demonstrated in prior lessons. A section at the end of the chapter describes other tutorials provided with the Collection Class Library that can help you with specific Collection Class Library techniques. Use this chapter if you are beginning to use the library and are unclear on some of the concepts described in earlier chapters of this section.

The lessons in this chapter demonstrate the following capabilities of the Collection Class Library:

- Defining a simple collection
- Adding, removing, and iterating over elements
- Changing the element type
- Changing the collection
- Changing the default implementation

Each lesson has the following format:

- *What the lesson covers*: What you will learn from the lesson.
- *Requirements*: What capabilities must be built into or added to the program.
- *Setup*: What files you will need from previous lessons.
- *Implementation*: Step-by-step instructions for implementing the program requirements. The implementation section includes the required code as well as detailed descriptions of each aspect of the implementation.
- *Source files*: Source file listings showing the contents of, or the order of declaration of functions within, individual source files. Where a source file is not changed from one lesson to the next, it is not listed a second time.
- *Running the program*: A description of what happens when you run the program, observations on the program's behavior, and guidance on optional ways of enhancing or changing the program.
- *What you have learned*: A summary of the Collection Class features that were covered by the lesson.

There are five lessons in this chapter. The following provides an overview of the characteristics of the program used in each lesson, and the Collection Class features the lesson demonstrates:

Lesson 1 A program that builds a collection of integer elements, and adds three elements to the collection. Nothing is done with the collection after these elements are added, and the program produces no output. This lesson demonstrates how to define the element and collection types with **`typedef`**s, how to instantiate a collection, how to add elements to a collection, and how to determine what Collection Class header file to include.

Lesson 2 An enhancement to Lesson 1 that implements a menu so that you can add, list, or remove items, show stock information, or exit the program. Not all these functions are fully implemented at this point. The lesson demonstrates how to iterate over a collection and how to remove elements from a collection.

Lesson 3 An enhancement to Lesson 2 that changes the element type from a built-in type to a class type. The lesson demonstrates how to construct a collection whose elements are of class type, how to determine what element type functions are required, and how to define those functions.

Lesson 4 In this lesson, you change Lesson 3 to use a different collection. The lesson demonstrates how to choose the correct collection for a given application, how to implement various element and key functions, how to use a cursor to iterate through elements with a given key, and how to count the number of elements with a given key.

Lesson 5 In this lesson, you change the implementation *variant* of the collection. This does not change the program's external behavior but in real applications changing an implementation variant can affect performance.

Preparing for the Lessons

To set up the lessons, create five directories beneath the same parent directory, and name them `lesson1` through `lesson5`. You will use these directories to store the files you create for each lesson.

Compiling the Lessons

To compile the lessons, use the following at the OS/2 command line:

```
icc -Iinclude_path1 ... -Iinclude_pathn -Fd -Ft -Tdp -Ti -B"/De " main.C
lib_path1\lib1 libpath2\lib2
```

where *`include_pathx`* is the path for included `.h` and `.hpp` files, and *`lib_pathx`* is the path for the required libraries. The libraries are `DDE4CC(I).LIB`, and for the `IString` class, `DDE4MUI(I).LIB`.

Note: The compiler creates a directory `tempinc` in the directory that is the current directory when the program is compiled. This directory is used by the Collection Class Library to place template files used to instantiate collections in your program. You can delete the files in `tempinc` and the directory itself after compilation.

If the compiler produces errors during compilation, check to make sure that you have specified the required library and that you have typed the source code in correctly. Some common errors are misplacing semicolons and failing to close braces or brackets.

Lesson 1: Defining a Simple Collection of Integers

In this lesson, you write a program that builds a very simple collection of integer elements and adds some elements to the collection. This lesson covers the following Collection Class topics:

- Using a **typedef** to define the element type
- Using a **typedef** to define the collection type
- Instantiating the collection
- Adding elements to the collection
- Specifying the Collection Class header file to include

Requirements

The collection must consist of elements of an integer type. The integer type is to be known as type `Bicycle`, so that later lessons can change the members of the type. The program adds three integers to the collection. Their values are unimportant. The collection is to be a bag.

Setup

Change to the `lesson1` directory, and use an editor to create and edit two files:

`bike.h` This file will contain declarations and typedefs for the element and collection types.

`main.C` This file will contain the **main()** function.

Implementation

The implementation should use **typedef**s to define the element and collection types, so that if the element or collection type changes later, the changes will be automatically reflected in any code that uses the **typedef**.

Defining the Element Type: Use a **typedef** to define a `Bicycle` as a synonym for an `int`. By using a **typedef**, you make it easier to change the element type later, without having to change anything outside the element's type (or class) definition:

```
// in bike.h
typedef int Bicycle;
```

Lesson 1: Defining a Simple Collection

Notes:

1. In a realistic C++ program using Collection Classes, you do not need to use a **typedef** to define the element type, because it is unlikely that you would switch from a built-in C++ type to a class type.
2. Unless otherwise indicated, you should enter each new block or line of code *below* any code you have already entered.

Defining the Collection Type: Use a **typedef** to define a collection type called `MyCollectionType`. The collection type refers to a bag collection whose elements are `Bicycles`. By using a **typedef**, you make it easier to change the collection type later, without having to change other parts of your code:

```
typedef IBag <Bicycle> MyCollectionType;
```

In this **typedef**, `IBag` is the default implementation for a bag, `Bicycle` is a template argument representing the element type, and `MyCollectionType` is the type name given to the type being defined (a bag of `Bicycle` elements).

Instantiating a Collection: Now that you have defined a **typedef** for both the element and the collection types, you can instantiate a collection with a type specifier and a name:

```
MyCollectionType MyCollection;
```

Place this definition at global scope so that all functions, not only the `main()` function defined in the next step, have access to the collection and its members. Functions other than `main()` are defined in subsequent lessons.

Adding Elements: You can use “Flat Collection Member Functions” in the *Open Class Library Reference* to determine what functions you need to use to manipulate elements of a collection. If you consult that chapter, you will find that the `add()` function is the function needed for this lesson. The syntax for `add()` is stated as:

```
void add (Element const& element);
```

For a collection named `MyCollection`, you can add elements using the following syntax:

```
MyCollection.add(aBicycle);
```

Where *`aBicycle`* is a `Bicycle` (in this case an integer). To add three elements, place code such as the following in `main.C`:

```
void main() {
   Bicycle a,b,c;
   a=458;
   b=12;
   c=365;
   MyCollection.add(a);
   MyCollection.add(b);
   MyCollection.add(c);
}
```

Include Files: Above any **typedef**s or instantiations that use Collection Classes, you must include the header file for any collection you are using. The chapter on bags in the *Open Class Library Reference* tells you what the header file is for the default implementation of a bag. You should add the following code to the start of `bike.h`, and include `bike.h` in `main.C`:

```
// in bike.h
#include <ibag.h>

// in main.C
#include "bike.h"
```

Source Files for Lesson 1

The files should now contain code similar to the following:

bike.h

```
#include <ibag.h>
typedef int Bicycle;
typedef IBag <Bicycle> MyCollectionType;
MyCollectionType MyCollection;
```

main.C

```
#include "bike.h"

void main() {
   Bicycle a,b,c;
   a=458;
   b=12;
   c=365;
   MyCollection.add(a);
   MyCollection.add(b);
   MyCollection.add(c);
}
```

Running the Program

Compile `main.C` and run the executable. The program does not produce any output, so it appears to do nothing. In fact, it adds three elements to a collection of integers. The collection is lost on program termination. The program is useless in practical terms, but does demonstrate some basic Collection Class concepts. Later lessons build on the code in this lesson, and provide greater functionality, including output of elements.

“Bag” in the *Open Class Library Reference* defines a number of element type functions as being required:

- Copy constructor
- Destructor

- Assignment
- Equality test (`operator==`)
- Ordering relation (`operator<`)

You did not have to define these functions in the above example, because for the built-in type `int`, and by extension the user-defined type `Bicycle`, these functions are already defined by the language.

What You Have Learned This lesson showed you how to define elements and collections using **`typedef`**s, how to instantiate a collection and elements, and how to add elements to that collection.

Lesson 2: Adding, Listing, and Removing Elements

The first lesson showed you how to create a simple collection and add three elements. This lesson moves the code for adding elements to a separate function, and implements functions for listing and removing elements as well. These functions are called from a main program that dispatches the appropriate function based on the user's choice of a menu option.

This lesson covers the following Collection Class topics:

- Iterating over a collection using iterators (`allElementsDo()`)
- Removing elements from a collection

Requirements

The code in the `main()` function must be replaced by a menu system that gives the user the following options:

1. Add an item
2. List all items
3. Remove an item
4. Show stock information
5. Exit program

Options 1 to 3 must be implemented through functions. Option 5 can be implemented by calling `exit()` or by exiting the scope of the menu selection loop and `main()`. You do not need to implement the function to show stock information in this lesson. Instead, you can implement a function that prints an error message stating that the function is not yet implemented. For all options except the exit option, after the appropriate function returns, the menu should be redisplayed and the user should be able to enter another selection.

Lesson 2: Adding, Listing, and Removing Elements

Setup

Copy the file bike.h from the lesson1 directory to the lesson2 directory, and then change your current directory to the lesson2 directory. You will also create two other files. The three files for this tutorial are:

bike.h	Contains the element and collection typedefs.
lesson.C	Contains functions for adding, removing, listing, and showing stock information on items.
main.C	Contains the main menu for the program.

Implementation

You need to replace the body of the main() function with the menu handling and function dispatching code. You will make use of I/O Stream input and output to implement the functions that add, list, or remove items. One advantage of using the I/O Stream classes instead of functions like printf() and scanf() is that, when the element type is changed, you can define input and output operators for the type, and the I/O Stream input and output functions will continue to work without change.

Including the iostream.h Header File: You should include iostream.h at the start of lesson.C so that you can use the cin, cout, and cerr streams that are predefined by the iostream class. You should also include the header file bike.h so that you can access the Bicycle class and associated functions.

```
#include <iostream.h>
#include "bike.h"
```

Adding Items: Before the definition of main(), define a function addItem() that requests user input for the item, then adds the item to the collection. The item is added using the add() function described in the first lesson. Here is one way to implement such a function:

```
// in lesson.C
void addItem() {
   Bicycle tbike;
   cout << "Enter item: ";
   cin >> tbike;
   while (cin.fail()) {
      cin.clear();
      cin.ignore(1000,'\n');
      cerr << "Input error, please re-enter: ";
      cin >> tbike;
   }
   MyCollection.add(tbike);
}
```

Note: You should also add a declaration for this and subsequent functions in main.C.

The function uses a temporary Bicycle object to contain the input until the element is copied into the collection. The function displays a prompt, reads input, and tests for valid input. The `while (cin.fail())` block clears any input errors and asks for input again. Once the element is successfully read from input, it is added to the collection.

Because `tbike` is actually an `int` in the current version, an `operator>>` is already defined for it. Later, when you change the `Bicycle` type to a user-defined class, you will have to add an `operator>>` for that class.

Listing Items: Before you can list all items, you must define a function that prints a single item. This function can then be invoked by the `allElementsDo()` member function of `MyCollection`. (📖 `allElementsDo()` is described in "allElementsDo" in the *Open Class Library Reference*.) Any function invoked by `allElementsDo()` must have a return type of `IBoolean`, and must have two arguments: a **`const`** reference to the argument and a pointer to void. The pointer to void is used to pass additional arguments to the applied function, if required by the function. For the printing function in this lesson you do not need to pass additional arguments, because the function does not use them. In such cases you pass a `void*` second argument:

```
// in lesson.C
IBoolean printItem (Bicycle const& bike, void* /* Not used */) {
   cout << bike << endl;
   return True;
}
```

The `printItem()` function should always return True because it should display the value of each element of the collection. If you wanted certain values of elements to cause printing to halt, you would have the function return False for any such element. A return value of False causes the `allElementsDo()` function to stop iterating over the collection.

Just as there was no need to define an input operator for `Bicycle`, there is no need to define an output operator either, as long as `Bicycle` represents an `int`.

Now define the function `listItems()` to call the `printItem()` function for each element of the collection. Use the `allElementsDo()` function for the collection, and use the `printItem()` function as argument. `allElementsDo()` then calls the function for every element of the collection.

```
// in lesson.C
void listItems() {
   MyCollection.allElementsDo( printItem );
}
```

Removing Items: To remove an element from a collection, you need to use the `remove()` member function. 📖 This function is described in "Flat Collection Member Functions" in the *Open Class Library Reference*. `remove()` returns True if the element was found in the collection and was removed, or it returns False if the element was not found in the collection. Your removal function should print an error if the element is not successfully removed. In the version below, the condition that determines whether removal was successful actually invokes the remove() function:

```
// in lesson.C
void removeItem() {
   Bicycle tbike;
   cout << "Enter item to remove: ";
   cin >> tbike;
   while (cin.fail()) {
      cin.clear();
      cin.ignore(1000,'\n');
      cerr << "Input error, please re-enter: ";
      cin >> tbike;
   }
   if (!MyCollection.remove(tbike))
      cerr << "Item not found!\n";
}
```

Showing Stock Information: For now, you can define this function to display an error message without changing the collection:

```
// in lesson.C
void showStock() {
   cerr << "Function not implemented yet!\n";
}
```

Main Menu: Finally, change the code in main() to display the menu items, accept input, and take appropriate action. Because this code will remain relatively unchanged for subsequent lessons, place it in a separate file, main.C, and include lesson.C before the code of main(). A possible version of main.C is shown below.

Source Files for Lesson 2

You should have two source files defined at this point. Their names and sample contents are:

main.C

```
#include <iostream.h>
#include <stdlib.h> // for use of exit() function
void addItem(), listItems(), showStock(), removeItem();

void main() {
   enum Choices { Add, List, Stock, Remove, Exit };
   int menuChoice=0;
   char* menu[5] = {"Add an item",
                    "List items",
                    "Show stock information",
                    "Remove an item",
                    "Exit" };
    while (menuChoice!=5) {
      cout << "\n\n\nSimple Stock Management System\n\n";
      for (int i=0;i<5;i++)
         cout << i+1 << ". " << menu[i] << '\n';
      cout << "\nEnter a selection (1-5): ";
      cin >> menuChoice;
```

Lesson 2: Adding, Listing, and Removing Elements

```
      while (cin.fail()) {
         // get input again if nonnumeric was entered
         cin.clear();
         cin.ignore(1000,'\n');
         cerr << "Enter a selection between 1 and 5!\n";
         cin >> menuChoice;
         }
      switch (menuChoice) {
         case 1: addItem();    break;
         case 2: listItems();  break;
         case 3: showStock();  break;
         case 4: removeItem(); break;
         case 5: exit(0);
         default: cerr << "Enter a selection between 1 and 5!\n";
      }
   }
}
```

lesson.C

```
// lesson.C
#include <iostream.h>
#include <ibag.h>
#include "bike.h"

void addItem() {
   Bicycle tbike;
   cout << "Enter item: ";
   cin >> tbike;
   while (cin.fail()) {
      cin.clear();
      cin.ignore(1000,'\n');
      cerr << "Input error, please re-enter: ";
      cin >> tbike;
   }
   MyCollection.add(tbike);
}

IBoolean printItem (Bicycle const& bike, void* /* Not used */) {
   cout << bike << endl;
   return True;
}

void listItems() {
   MyCollection.allElementsDo( printItem );
}

void removeItem() {
   Bicycle tbike;
   cout << "Enter item to remove: ";
   cin >> tbike;
   while (cin.fail()) {
      cin.clear();
      cin.ignore(1000,'\n');
      cerr << "Input error, please re-enter: ";
      cin >> tbike;
   }
   if (!MyCollection.remove(tbike))
      cerr << "Item not found!\n";
}

void showStock() {
   cerr << "Function not implemented yet!\n";
}
```

Running the Program

Compile `main.C` and `lesson.C`, link them, and run the program. You can enter elements into the collection, list the elements, remove them, or exit from the program. If you select the option to display stock information, an error message is displayed and no action is taken.

Elements appear to be ordered: If you enter more than one integer into the collection, and then list the collection's elements, you may find that the collection has been sorted from the smallest to the largest element. Do not rely on this ordering relation, because a `Bag` is an unordered, unsorted collection, and changes to your code or to the Collection Class Library could change the order in which elements are accessed.

Multiple equal elements are supported: If you add the number 7 to the collection three times and list the items, the number 7 appears three times. If you then remove the number 7 once, the number 7 still appears twice. A bag supports multiple equal elements.

What You Have Learned

This lesson showed you how to use the `allElementsDo()` function to iterate over elements of a collection, and how to provide a function to `allElementsDo()` that is called for each iterated element. The lesson also demonstrated how to use the `remove()` function to remove elements from a collection.

Lesson 3: Changing the Element Type

Now that you have a working program that allows you to add, list, or remove elements from a collection, you are ready to change the element type to something more complex than an integer.

This lesson covers the following Collection Class topics:

- Defining an element type as a class
- Determining what element type functions are required
- Defining those element type functions

Requirements

The element type must be changed from the built-in integer type to a class type with the following data members:

- A string representing the manufacturer or make of the bicycle
- A string representing the model of the bicycle
- An integer representing the type of bicycle: racing, touring, or mountain bike
- An integer representing the price of the bicycle

Lesson 3: Changing the Element Type

Setup

Copy the files bike.h, lesson.C, and main.C from the lesson2 directory to the lesson3 directory, and then change your current directory to the lesson3 directory. Use an editor to modify these files, and to create a new file bike.C, which will contain function definitions for functions declared in bike.h.

Implementation

First move the **typedef** definition for the collection and the **#include** statement for ibag.h from bike.h to lesson.C, where they are actually made use of.

You can use the IString class to handle the strings for make and model. This class includes operators for element equality, ordering relation, and addition (concatenation), all of which will be used in this or later lessons.

Defining the Element Type: In keeping with good object-oriented programming practice, you should separate the member function definitions from the class definition, by placing the class definition in bike.h and the definitions of member functions in bike.C. You should compile each .C file separately, and link them together.

Class Data Members: The following code defines the data members of Bicycle. You should replace the **typedef** for the element with the declaration for class Bicycle. Two header files are also included because they are required by members of the class. Place the following code in bike.h.

```
#include <istring.hpp>  // access to IString class
#include <iostream.h> // access to iostream class

class Bicycle {
   public:
      IString Make;
      IString Model;
      int Type;
      int Price;
// ... Member functions to be declared later and defined in bike.C
};
```

The following code defines an enumerator (used to determine the type of bicycle) and an array of IString objects (used to display the types of bicycle). Place it in bike.C:

```
enum bikeTypes { Racing, Touring, MountainBike };
IString btype[3]={ "Racing", "Touring", "Mountain Bike"};
```

Selecting What Element Type Functions to Implement: When you implement the element type as a user-defined type (a class), you must define certain element functions, and in some cases key-type functions, for that element. These functions are used by Collection Class functions to locate, add, copy, remove, sort, or order elements within their collection, and to determine whether two elements of a collection are equal. For example, you may need to define element equality through an `operator==`, so that Collection Class functions can determine whether an element you try to add to the collection is identical to an element already present in the collection. Provided you use the correct return type and calling arguments, there is no right or wrong way to code many of these functions. An equality function for elements consisting of two `int` data members, for example, could return True (meaning that two elements are equal) if the *difference* between the two data members is the same for both elements. In this case, the objects (3,8) and (4,9) would be equal.

To determine what element and key-type functions you need to implement for a given collection, you should consult the appropriate collection's chapter in the *Open Class Library Reference*. For this lesson, the collection is a bag. When you are first developing a program, you should use the default implementation of the collection, which is always the first implementation variant listed under the chapter's "Template Arguments and Required Functions" section. For each implementation variant, a list of required functions is provided, and you must either implement these functions for your element class, or determine that they are automatically generated by the compiler. In the case of the default implementation of a Bag, the following required functions are shown, under the heading "Element Type":

- Copy constructor
- Destructor
- Assignment
- Equality test
- Ordering relation

For this lesson, you also need to implement input and output operators and a default constructor (used by the input operator and other functions).

Default Constructor: The default constructor should initialize all data members to blank strings or zero integers:

```
// in bike.h, within class declaration
Bicycle() : Make(""), Model(""), Type(0), Price(0) {}
```

Assignment Operator and Destructor: There is no need to define these explicitly. The compiler generates a default assignment operator and destructor that are suitable for the program.

Copy Constructor: This function is used by the Collection Classes and by the input operator. Declare and define it as follows:

```
// in bike.h:
Bicycle(IString mk, IString md, int tp, int pr) :
   Make(mk), Model(md), Type(tp), Price(pr) {}
```

Equality Test: The equality test (operator==) should return True if two bicycles have the same make and model, and False if not:

```
// in bike.h:
IBoolean operator== (Bicycle const& b) const;

// in bike.C:
IBoolean Bicycle::operator== (Bicycle const& b) const
   { return ((Model==b.Model) && (Make==b.Make)); }
```

Ordering Relation: The ordering relation (operator<) should indicate whether the first bicycle would appear before or after the second bicycle in an alphabetically sorted list:

```
// in bike.h:
IBoolean operator< (Bicycle const& b) const;

// in bike.C:
IBoolean Bicycle::operator< (Bicycle const& b) const
   { return ((Make<b.Make) || (Make==b.Make && Model<b.Model)); }
```

You can use the < and == operators for IString objects because they are defined for the IString class to indicate alphanumeric sorting order.

Input Operator: This operator is required by the addItem() and removeItem() functions defined previously. Both this and the output operator are declared *outside* the class definition, at the bottom of bike.h, and they are defined in bike.C. The input operator stores the alphanumeric data members of Bicycle in char arrays to avoid the overhead of constructing temporary IString objects.

```
// in bike.h:
istream& operator>> (istream& is, Bicycle& bike);

// in bike.C:
istream& operator>> (istream& is, Bicycle& bike) {
   char make[40], model[40];
   char typeChoice;
   float price;
   int type=-1;
   cin.ignore(1,'\n'); // ignore linefeed from previous input
   cout << "\nManufacturer: ";
   cin.getline(make, 40, '\n');
   cout << "Model: ";
   cin.getline(model, 40, '\n');
   while (type == -1) {
      cout << "Racing, Touring, or Mountain Bike (R/T/M): ";
      while (cin.fail()) {
         cin.clear();
         cin.ignore(1000,'\n');
         cerr << "Racing, Touring, or Mountain Bike (R/T/M): ";
         cin >> typeChoice;
      }
      switch (typeChoice) {
         case 'r':
         case 'R': { type=Racing; break; }
         case 't':
         case 'T': { type=Touring; break; }
         case 'm':
         case 'M': { type=MountainBike; break; }
         default:  { cerr << "Incorrect type, please re-enter\n"; }
      }
   }
   cout << "Price ($$.$$): ";
   cin >> price;
   while (cin.fail()) {
      cin.clear();
      cin.ignore(1000,'\n');
      cerr << "Enter a numeric value: ";
      cin >> price;
      }
   price*=100;
   bike=Bicycle(make,model,type,price);
   return is;
}
```

Output Operator: The output operator is required by the `listItems()` function, and may later be required by other functions. It should display the make, model, type, and price of a bicycle:

```
// in bike.h:
ostream& operator<< (ostream& os, Bicycle bike);

// in bike.C:
ostream& operator<< (ostream& os, Bicycle bike) {
   return os << bike.Make
             << "\t" << bike.Model
             << "\t" << btype[bike.Type]
             << "\t" << float(bike.Price)/100;
}
```

Source Files for Lesson 3

The program should now be placed in the following files. Some function bodies have been replaced with ellipses for brevity. `main.C` remains unchanged and is not shown.

lesson.C

```
// lesson.C
#include <iostream.h>
#include <ibag.h>
#include "bike.h"
typedef IBag<Bicycle> MyCollectionType;
MyCollectionType MyCollection;

void addItem() { /* ... */ }
IBoolean printItem (Bicycle const& bike, void* /* Not used */)
   { /* ... */ }
void listItems() { /* ... */ }
void removeItem() { /* ... */ }
void showStock() { /* ... */ }
```

bike.h

```
#include <istring.hpp>  // access to IString class
#include <iostream.h> // access to iostream class

class Bicycle {
   public:
   IString Make;
   IString Model;
   int Type;
   int Price;
   Bicycle() : Make(""), Model(""), Type(0), Price(0) {}
   Bicycle(IString mk, IString md, int tp, int pr) :
   Make(mk), Model(md), Type(tp), Price(pr) {}
   IBoolean operator== (Bicycle const& b) const;
   IBoolean operator< (Bicycle const& b) const;
};
istream& operator>> (istream& is, Bicycle& bike);
ostream& operator<< (ostream& os, Bicycle bike);
```

bike.C

```
#include <istring.hpp>
#include "bike.h"
enum bikeTypes { Racing, Touring, MountainBike };
IString btype[3]={ "Racing", "Touring", "Mountain Bike"};

IBoolean Bicycle::operator== (Bicycle const& b) const
       { return ((Model==b.Model) && (Make==b.Make)); }

IBoolean Bicycle::operator< (Bicycle const& b) const
       { return ((Make<b.Make) || (Make==b.Make && Model<b.Model)); }

istream& operator>> (istream& is, Bicycle& bike) {
       char make[40], model[40];
       char typeChoice;
       float price;
       int type=-1;
       cin.ignore(1,'\n'); // ignore linefeed from previous input
       cout << "\nManufacturer: ";
       cin.getline(make, 40, '\n');
       cout << "Model: ";
       cin.getline(model, 40, '\n');
       while (type == -1) {
          cout << "Racing, Touring, or Mountain Bike (R/T/M): ";
          cin >> typeChoice;
          while (cin.fail()) {
             cin.clear();
             cin.ignore(1000,'\n');
             cerr << "Racing, Touring, or Mountain Bike (R/T/M): ";
             cin >> typeChoice;
          }
          switch (typeChoice) {
             case 'r':
             case 'R': { type=Racing; break; }
             case 't':
             case 'T': { type=Touring; break; }
             case 'm':
             case 'M': { type=MountainBike; break; }
             default:  { cerr << "Incorrect type, please re-enter\n"; }
          }
       }
       cout << "Price ($$.$$): ";
       cin >> price;
       while (cin.fail()) {
          cin.clear();
          cin.ignore(1000,'\n');
          cerr << "Enter a numeric value: ";
          cin >> price;
       }
       price*=100;
       bike=Bicycle(make,model,type,price);
       return is;
    }
```

```
ostream& operator<< (ostream& os, Bicycle bike) {
    return os << bike.Make
              << "\t" << bike.Model
              << "\t" << btype[bike.Type]
              << "\t" << float(bike.Price)/100;
 }
```

Running the Program

Compile and link `bike.C`, `main.C` and `lesson.C`, and then run the program.

If you add two bicycles with the same make and model, but different types or prices, the second bicycle's entry will be identical to the first when the bicycles are listed. The reason is that element equality is defined only in terms of the make and model. When you add what the collection considers to be an equal element, the existing element is duplicated by the `add()` function.

When you remove an item, the input operator asks you to enter all fields for the item to remove. Again, because element equality is defined only for the make and model fields, the information you provide for bicycle type and price is not used in determining which element to remove. If you define a bicycle:

```
Smithson    37Q    Racing   $270.00
```

You can remove that bicycle's entry by removing:

```
Smithson    37Q    Mountain Bike   $399.99
```

These limitations will be corrected in the next lesson.

What You Have Learned

In this lesson, you moved from using built-in types as elements of a collection to using user-defined or class types. When you create a collection using class-type elements, you must define certain element functions. This lesson showed you how to determine what element functions are required, and how to implement them.

Lesson 4: Changing the Collection

When you design an actual application using the Collection Class Library, you should choose the collection best suited to your program at the design stage. Nevertheless, requirements may change, and if you have followed the techniques used in this lesson such as specifying the collection type with a **typedef**, you can change the collection type without having to rewrite the entire application. Only minor changes are required to existing code, and a few simple element or key-type functions may need to be added or changed.

This section illustrates the following Collection Class concepts:

- Selecting the correct collection type
- Implementing a key
- Defining key access

Lesson 4: Changing the Collection

- Defining key equality
- Defining a key hash
- Using a cursor to iterate through elements with a given key
- Counting the number of elements of a given key

Requirements

The program should be changed so that two bicycles of the same model and make can have different type and price information. When users asks to delete a bicycle, they should not have to enter the bicycle and price information; instead, a list of all bicycles of the specified make and model should be displayed, and the user should be able to select which bicycle to remove from the collection. The `showStock()` function should also be implemented, so that it shows the number of a given make and model of bicycle currently in the collection.

Setup

Copy the files `bike.h`, `bike.C`, `lesson.C`, and `main.C` from the `lesson3` directory to the `lesson4` directory, and then change your current directory to the `lesson4` directory. Use an editor to modify the files as described below.

Implementation

The collection must have the following characteristics:

Key access, so that an element can be accessed using only its make and model information (for the listing and removing functions)

No element order, because order is not specified as a requirement

Multiple elements with the same key, so that several bicycles of the same make and model can be present in the collection

Element equality, so that elements with the same make and model can have different price and type information

You can use Figure 7 on page 81 to determine what collection best meets the requirements listed above. Begin by applying one requirement to the figure to narrow down the number of possible collections. Apply a second requirement to the remainder, and continue until you have found all valid collections. In this example, there is one valid collection, selected as follows:

- Elements have a key (the make and model). This means that any of the following collections may be a candidate:
  - Map
  - Relation
  - Sorted map
  - Sorted relation
  - Key set
  - Key bag

  - Key sorted set
  - Key sorted bag

- The order of elements is not important. This means that all sorted collections can be removed from the list above, leaving:
  - Map
  - Relation
  - Key set
  - Key bag
- Multiple elements may have the same key. This leaves relation and key bag.
- Element equality is required, so that individual elements with the same key can be distinguished. This leaves relation.

A relation differs from a bag in that it is instantiated using a key type as well as the element type, and requires the following additional functions:

Element type: Key access

Key type: Equality test and hash function.

These functions are defined below.

Changing the Collection Type Definition: Before you redefine the functions in `lesson.C`, you need to change the include file and **`typedef`** for the collection type so that they use relation instead of bag:

```
// lesson.C
#include <irel.h> // was ibag.h
//...
typedef IRelation<Bicycle,IString> MyCollectionType;
// was typedef IBag<Bicycle> MyCollectionType;
```

Notice that `IRelation` takes two template arguments, an element type and a key type. All collections that have a key must be defined with a template argument for key type as well as one for element type.

Ordering Relation: A relation does not require an operator for ordering relation (`operator<`). You defined this operator when the collection was implemented as a bag. You should comment it out or remove it for this implementation. This function is declared in `bike.h` and defined in `bike.C`.

Implementing a Key: The key consists of the make and model of the bicycle. You can use an `IString` to implement the key. Because the return value of the `key()` function must be a **`const`** reference, and because the `key()` function cannot change the element, the key must be determined before the `key()` function is called. The logical place to do this is in the element constructor (in `bike.h`), because the overhead of generating the key only occurs once per element. You can add a key data member to the collection, and have it initialized when the copy constructor is called. In the example below, the key is named MMKey (which stands for Make/Model Key):

```
// in bike.h:
class Bicycle {
      IString MMKey; // add a private data member for the key
   public:
      // public data members and member functions
      Bicycle(IString mk, IString md, int tp, int pr);
      // ...
};

// in bike.C:
Bicycle::Bicycle(IString mk, IString md, int tp, int pr) :
        Make(mk), Model(md), Type(tp), Price(pr),
        MMKey(mk+md) {}
```

Defining Key Access: The key access function must be defined *outside* of the element class. It has one argument, whose type is the element type. The key access function must call a member function that returns the key, in this case a function named getKey(). (The actual name does not matter.) The member function accesses the private data member MMKey.

```
// in bike.h:
class Bicycle {
      IString MMKey;
   public: // ... data members and member functions
      IString const& getKey() const;
   };

inline IString const& key (Bicycle const& bike)
   { return bike.getKey(); }

// in bike.C:
IString const& Bicycle::getKey() const { return MMKey; }
```

The key access function must be declared with the name key(), with a **const** reference to the key as its return value, and a **const** reference to the element as its argument.

Equality Test: Equality for elements should be defined such that the key (that is, the make and model), the type, and the price are the same for two bicycles. The operator== function in bike.C can be redefined as follows:

```
 IBoolean Bicycle::operator== (Bicycle const&b) const {
     return (MMKey==b.MMKey && Type==b.Type && Price==b.Price);
     }
```

Key Hash Function: The hash function provides a shortcut for Collection Class search functions to find matches to a key. The search functions first call the hash function on a key for which they need to locate an element. They use the hash value returned to look for matches to that hash in a hash table. They then use the full key to determine which of the hash function's matches have the correct key. The hash key-type function is not a member function of the element's class. It is called by the searching function, with a key argument (the key on which to derive the hash) and an unsigned long (the maximum hash value). The return value is the hash, and it cannot exceed the maximum hash value. The hash function should be defined in `lesson.C` and must have the following return type and parameters:

```
unsigned long hash ( IString const& keyName,    unsigned long hashInput );
```

You can define the hash using the hashing function provided in `istdops.h` for `char*` values:

```
unsigned long hash (IString const &aKey, unsigned long hashInput) {
    return hash( (const char*)aKey, hashInput);
    }
```

Using Cursors to Remove Items: A Collection Class cursor (not related to the cursor used to move about a cursor screen) is a reference to an element in a collection. For an overview of cursors, see “Cursors” on page 98.

The `removeItem()` function must be redefined so that it requests the make and model of bicycle to remove, lists all matching bicycles, and lets the user choose which match to remove. Once matching bicycles have been displayed, a cursor can be used to locate the bicycle the user wishes to delete. The cursor is defined as follows, immediately after the collection `MyCollection` is declared, in `lesson.C`:

```
MyCollectionType::Cursor thisOne (MyCollection);
```

After the user enters a make and model to search for, the `removeItem()` function should iterate through all elements that match the key, by using `locateElementWithKey()` to find the first matching element, and `locateNextElementWithKey()` to find all subsequent matching elements. Both these functions require a cursor as their second argument, and the cursor points to the located element when the functions return. The first part of `removeItem()` can be redefined as follows:

```
void removeItem() {
   Bicycle tbike;
   int choice, cursct=1;
   cout << "\nRemove an item";
   cin >> tbike;
   if (MyCollection.numberOfElementsWithKey(tbike.getKey()) > 0) {
      MyCollection.locateElementWithKey(tbike.getKey(), thisOne);
      cout << cursct << ". " << MyCollection.elementAt(thisOne) << endl;
      for ( cursct=2;
            MyCollection.locateNextElementWithKey(
                             tbike.getKey(), thisOne);
            cursct++)
            {  cout << cursct << ". "
                    << MyCollection.elementAt(thisOne) << endl; }
   //... Remainder to be defined later
   }
```

In the above fragment, the user is asked for a bicycle make and model to remove. If any elements match the make and model (this is determined by testing the numberOfElementsWithKey() function for a nonzero return), all such elements are located by key. The locateElementWithKey() function sets its cursor to point to the first matching element, and the locateNextElementWithKey() function advances the cursor from the current match to the next match in the collection. The elements are accessed for output using the elementAt() function, which returns a reference to the element pointed to by the cursor argument.

Once the matching elements have been displayed with a number beside each one, the program should ask the user to enter a number matching the number of the element to remove. The matching elements can then be iterated over again until the number of elements iterated over matches the user's selection, and the element pointed to by the cursor is then deleted. The following code excerpt is the second part of the removeItem() function:

```
// Insert this at "...Remainder to be defined later" in removeItem().
    cout << "\nEnter item to remove, or 0 to return: ";
    cin >> choice;
    if (choice<=0 || choice > cursct) return;
    MyCollection.locateElementWithKey(tbike.getKey(),thisOne);
                 // locate the first matching element again
    for ( cursct=2;
          cursct<=choice &&
          MyCollection.locateNextElementWithKey    // check for valid
                    (tbike.getKey(), thisOne);  // next match
          cursct++)
          ; // null loop - header contains the code to be executed
    MyCollection.removeAt(thisOne);
    }
 else
    cerr << "No bicycles of this make and model were found.\n";

// The closing brace below was already part of removeItem().
// Do not duplicate it.
 }
```

Note: The locateNextElementWithKey() function invalidates the cursor if it cannot find a next element with the key provided. An invalidated cursor does not point to any element of the collection. Some flat collection member functions that use cursors require that the cursor be valid (locateNextElementWithKey() is one such function). Before you use a cursor with such a function, you need to validate the cursor by using a function that takes a cursor as argument but does not require a valid cursor on entry. locateElementWithKey() is one such function.

In both excerpts of removeItem() above, the elements with matching keys are iterated over by code in the header of the loop. In the second case, the loop has no body. You can use this coding style because all the locate... functions have a return type of IBoolean, which can be used in condition tests such as those in loop control expressions.

Showing Stock Information: showStock() must be rewritten so that, for a given make and model, it displays the number of matching elements in the collection. The numberOfElementsWithKey() function can be used:

```
void showStock() {
   Bicycle tbike;
   int count;
   cout << "Stock information for a model";
   cin >> tbike;
   count=MyCollection.numberOfElementsWithKey(tbike.getKey());
   if (count!=1)
      cout << "Currently there are " << count << " bicycles ";
   else
      cout << "Currently there is 1 bicycle ";
   cout << "of this make and model in stock." << endl;
   }
```

Changing the Input Operator and addItem(): As the program now stands, the input operator requests input for all data members of `Bicycle`, including type and price information. This means that, when you select an item to remove or to show stock information on, you must specify type and price information even though this information is ignored. Therefore you need to move the request for type and price information out of the `operator>>` definition in `bike.C` and into `addItem()`, so that the user only needs to enter type and price information when an item is being added to the collection. You also need to add the enumeration `bikeTypes` to `lesson.C` so that `addItem()` has access to them.

See the "Source Files" section below for the changes required to `addItem()` and `operator>>`.

Source Files for Lesson 4

The main program in `main.C` has not been changed. The following excerpts show the layout of code between `lesson.C` and `bike.h`. Function bodies that remain unchanged from the preceding lesson have been replaced by ellipses.

bike.h

```
#include <istring.hpp>   // access to IString class
#include <iostream.h>  // access to iostream class

class Bicycle {
      IString MMKey;
   public:
      IString Make;
      IString Model;
      int Type;
      int Price;
      Bicycle();
      Bicycle(IString mk, IString md, int tp, int pr);
      IBoolean operator== (Bicycle const& b) const;
//    IBoolean operator< (Bicycle const& b) const;
      IString const& getKey() const;
   };

inline IString const& key (Bicycle const& bike)
   { return bike.getKey(); }

istream& operator>> (istream& is, Bicycle& bike);
ostream& operator<< (ostream& os, Bicycle bike);
```

Lesson 4: Changing the Collection

bike.C

```
#include <istring.hpp>
#include "bike.h"
enum bikeTypes { Racing, Touring, MountainBike };
IString btype[3]={ "Racing", "Touring", "Mountain Bike"};

Bicycle::Bicycle() : Make(""), Model(""), Type(0), Price(0) {}
Bicycle::Bicycle(IString mk, IString md, int tp, int pr) {
    Make=mk;
    Model=md;
    Type=tp;
    Price=pr;
    MMKey=Make+Model;
    }
// Comment out the ordering relation operator
// IBoolean Bicycle::operator< (Bicycle const& b) const
//    { return ((Make<b.Make) || (Make==b.Make && Model<b.Model)); }
IBoolean Bicycle::operator== (Bicycle const&b) const {
  return (MMKey==b.MMKey && Type==b.Type && Price==b.Price);
  }
IString const& Bicycle::getKey() const { return MMKey; }

istream& operator>> (istream& is, Bicycle& bike) {
    char make[40], model[40];
    char typeChoice;
    float price=0;
    int type=-1;
    cin.ignore(1,'\n'); // ignore linefeed from previous input
    cout << "\nManufacturer: ";
    cin.getline(make, 40, '\n');
    cout << "Model: ";
    cin.getline(model, 40, '\n');
    bike=Bicycle(make,model,type,price);
    return is;
 }

ostream& operator<< (ostream& os, Bicycle bike) {/* ... */} // unchanged
```

lesson.C

```
// lesson.C
#include <iostream.h>
#include <irel.h> // was ibag.h
#include "bike.h"
enum bikeTypes { Racing, Touring, MountainBike };
typedef IRelation<Bicycle,IString> MyCollectionType;

MyCollectionType MyCollection;
MyCollectionType::Cursor thisOne (MyCollection);

IBoolean printItem (Bicycle const& bike, void* /* Not used */)
   { /* ... */ }

void addItem() {
   Bicycle tbike;
   char typeChoice;
   float price;
   int type=-1;
   cout << "Enter item: ";
   cin >> tbike;
   while (type == -1) {
      cout << "Racing, Touring, or Mountain Bike (R/T/M):";
      cin >> typeChoice;
      while (cin.fail()) {
         cin.clear();
         cin.ignore(1000,'\n');
         cerr << "Racing, Touring, or Mountain Bike (R/T/M): ";
         cin >> typeChoice;
         }
      switch (typeChoice) {
         case 'r':
         case 'R': { type=Racing; break; }
         case 't':
         case 'T': { type=Touring; break; }
         case 'm':
         case 'M': { type=MountainBike; break; }
         default:  { cerr << "Incorrect type, please re-enter\n"; }
      }
   }
   cout << "Price ($$.$$): ";
   cin >> price;
   price*=100;
   tbike.Type=type;
   tbike.Price=price;
   MyCollection.add(tbike);
}

void listItems() {/* ... */ }
void removeItem() {
   Bicycle tbike;
   int choice, cursct=1;
   cout << "\nRemove an item";
   cin >> tbike;
   if (MyCollection.numberOfElementsWithKey(tbike.getKey()) > 0) {
      MyCollection.locateElementWithKey(tbike.getKey(), thisOne);
      cout << cursct << ". " << MyCollection.elementAt(thisOne) << '\n';
      for ( cursct=2;
            MyCollection.locateNextElementWithKey(
                              tbike.getKey(), thisOne);
            cursct++)
            {  cout << cursct << ". "
                    << MyCollection.elementAt(thisOne) << '\n'; }
      cout << "\nEnter item to remove, or 0 to return: ";
      cin >> choice;
      if (choice<=0 || choice > cursct) return;
      MyCollection.locateElementWithKey(tbike.getKey(),thisOne);
                   // locate the first matching element again
      for ( cursct=2;
            cursct<=choice &&
            MyCollection.locateNextElementWithKey      // check for valid
                         (tbike.getKey(), thisOne);  // next match
            cursct++)
            ; // null loop - header contains the code to be executed
      MyCollection.removeAt(thisOne);
      }
   else
      cerr << "No bicycles of this make and model were found.\n";
   }

void showStock() {
   Bicycle tbike;
   int count;
   cout << "Stock information for a model";
   cin >> tbike;
```

```
      count=MyCollection.numberOfElementsWithKey(tbike.getKey());
      if (count!=1)
         cout << "Currently there are " << count << " bicycles ";
      else
         cout << "Currently there is 1 bicycle ";
      cout << " of this make and model in stock." << endl;
      }

   unsigned long hash (IString const &aKey, unsigned long hashInput) {
       return hash( (const char*)aKey, hashInput);
       }
```

Running the Program

You can enter multiple bicycles of the same make and model, with different price or type information, and all such models will appear when you select the “List items” option. When you ask for stock information, the program displays the number of elements in the collection that match the make and model information you specify. When you remove an item, the program asks you for the make and model, displays a list of matching items, and lets you specify which item to remove. The program removes that item.

What You Have Learned

The Collection Class Library offers a wide range of collections with different characteristics. In this lesson, you learned how to select an appropriate collection based on the characteristics of the data being placed in the collection and on the intended uses of the data. Many Collection Classes are accessed or sorted using a key, and you learned how to define key access, equality, and hash functions, and how to iterate through a key collection using a key cursor.

Lesson 5: Changing the Implementation Variant

You should pursue changing the default implementation to an implementation variant only after the program is functionally complete and has been fully debugged. The purpose of changing to a nondefault implementation variant is to improve performance. This lesson shows you how to change the code defined in “Lesson 3: Changing the Element Type” on page 161 so that it is functionally equivalent, but uses `IBagOnSortedDilutedSequence` rather than `IBag`. The lesson assumes that you have done some analysis of your code, and have determined that this implementation variant may provide better performance. In the case of a full-fledged application, once you change the implementation variant, you should compile the program and time it against the original implementation to determine whether there is a worthwhile gain in performance.

This section illustrates the following Collection Class concepts:

- Changing the implementation variant header file
- Changing the implementation variant template and template arguments
- Determining what functions are required by the implementation variant

Lesson 5: Changing the Implementation Variant

Requirements

The only implementation variant for a relation is the variant that allows you to use a generic operations class.

If the collection were still a bag, a number of implementation variants would be available. In the third lesson, you used the default implementation variant for a bag. Other implementation variants are:

- Bag on B* key sorted set
- Bag on sorted linked sequence
- Bag on sorted tabular sequence
- Bag on sorted diluted sequence
- Bag on hash key set

For this lesson, you will use the code from the third lesson as a starting point, and change the default Bag implementation.

Setup

Copy the files `bike.h`, `bike.C`, `lesson.C`, and `main.C` from the `lesson3` directory (*not* the `lesson4` directory) to the `lesson5` directory, and then change your current directory to the `lesson5` directory. Use an editor to modify the files as described below.

Implementation

To change the default implementation of a collection to another implementation variant, you need to change the Collection Class file that you include, the collection **`typedef`**, and potentially the element and key functions.

Implementation Variant Header Files: To determine the correct header file to include, consult the "Class Implementation Variants" section of the chapter on Bag in the *Open Class Library Reference*. The header file to include for `IBagOnSortedDilutedSequence` is shown as `ibagsds.h`. You therefore change the header file to include as follows:

```
// in lesson.C
// old:
/* #include <ibag.h> */
// new:
#include <ibagsds.h>
```

Templates for Implementation Variants: To determine the correct template to instantiate for the collection **`typedef`**, see the implementation variant in the appropriate collection chapter. In this case, you would look for "Bag on Sorted Diluted Sequence" in 📖 "Bag" in the *Open Class Library Reference*. The collection is shown there as:

```
IBagOnSortedDilutedSequence  <Element>
IGBagOnSortedDilutedSequence <Element, ECOps>
```

Lesson 5: Changing the Implementation Variant

Because you are not defining a generic operations class, you need to use the first implementation variant. You therefore change the **typedef** for the collection as follows:

```
// old: typedef IBag <Bicycle> MyCollectionType;
// new:
typedef IBagOnSortedDilutedSequence <Bicycle> MyCollectionType;
```

Element Type Functions: To determine the required element type functions, see the "Element Type" section for the implementation variant. In the case of `IBagOnSortedDilutedSequence`, the only element type function listed that was not listed for a Bag is the default constructor, which is already defined in `Bicycle` for other reasons. If other functions are required for a given implementation variant you choose to use in an application, use the information on implementing a hash function in Lesson 4 for hints on where to place and how to code such functions.

No further changes are required. For this lesson, the only implementation variant that would require additional element type functions is `IBagOnHashKeySet`, and the required additional function is a hash function, which is already described in "Lesson 4: Changing the Collection" on page 167.

Running the Program

The program should have the same behavior, for a given set of inputs, as the program from "Lesson 3: Changing the Element Type" on page 161. In a complex application, a change in performance might occur, but in all cases the behavior of a correctly coded program should be identical for different implementation variants of the same collection class.

What You Have Learned

Once a C++ program using the Collection Classes is functionally complete and error-free, you can focus on performance. The key to good performance of Collection Classes programs is to select the appropriate implementation variant of a given collection. Although this lesson did not explain which implementation variant to choose (since this is largely dependent on the class type being used in the collection and on other factors beyond the scope of the lessons), it showed you how to change the implementation variant once the appropriate variant has been selected. 🕮 See "Features of Provided Implementation Variants" on page 130 for guidance on what implementation variants to select for a given application.

Errors When Compiling or Running the Lessons

If you code the programs in this chapter exactly as shown, they should compile successfully, and should run without any errors except those related to incorrect user input. Check your code for typographical mistakes or incorrectly placed code if you get compiler errors.

If you implement element, key, input, or output functions in different ways than those indicated, and your program does not compile successfully, or compiles but ends with an exception message when run, you can use Chapter 15, "Solving Problems in the Collection Class Library" on page 181 to determine the cause. You can also use Chapter 15 to find errors related to using a different collection or implementation variant from those specified in the lessons.

Other Tutorials

The Collection Class Library tutorials provided with VisualAge C++ can help you to learn the concepts of the Collection Classes They are presented in the same order as the Collection Class Library topics in this book. You should be familiar with the information in the first three chapters of Part 3 before beginning the tutorials.

Using the Default Classes

When you are learning to use a particular collection, you should first use the default class of that collection, so that you can gain a fundamental understanding of the collection before you approach the implementation variants of the collection.

You need to understand the topics covered in the following sections to successfully complete the tutorials:

Tutorial 1 Use of default implementations ("Instantiation and Object Definition" on page 95)

Tutorial 2 Adding, removing and replacing elements in a collection ("Adding, Removing, and Replacing Elements" on page 96)

Tutorial 3 Use of a cursor, locating and accessing elements, and the use of iterators ("Cursors" on page 98, "Using Cursors for Locating and Accessing Elements" on page 100, "Iterating over Collections" on page 101)

Tutorial 4 Use of exceptions (Chapter 13, "Exception Handling" on page 145)

After completing the above tutorials, you should be acquainted with the basic features of the Collection Class Library. For a more thorough understanding of the library, use the tutorials described below.

Other Tutorials

Advanced Use

If you want to understand more advanced uses of the classes, use tutorials 5 and 6. You need to understand the topics covered in the following sections to successfully complete the tutorials:

Tutorial 5 Exchanging implementation variants (Chapter 10, “Tailoring a Collection Implementation” on page 125)

Tutorial 6 Using abstract base classes to write polymorphic functions (Chapter 11, “Polymorphic Use of Collections” on page 139)

Source Files for the Tutorials

Each tutorial's files are stored in a separate directory. The tutorials are contained in subdirectories with the name `...\tutorial\iclcc\tutor?` where `?` corresponds to the number of the tutorial (1-6). Every directory contains the following files:

`tutor?.rea`	Read this to understand the purpose of the tutorial.
`tutor?.txt`	Instructions to follow.
`tutor?.mak`	Prepared makefile to compile the example.
`solution`	Directory containing a possible solution.

You will find prepared `.c` and `.h` files, where certain parts are missing. The objective of the tutorials is to apply the information you have learned about the Collection Class Library by adding the missing parts for each file. Complete the prepared files following the instructions in `instruct.txt`. You can compare your solutions to the solutions directory.

15

Solving Problems in the Collection Class Library

This chapter helps you solve problems that you may encounter when you use the Collection Class Library. The following table provides a short summary of each problem, and directs you to a section containing hints for a solution.

Problem Area	Problem Effect	Page
Cursor Usage	Unexpected results when using cursors	182
Element Functions and Key-Type Functions	Error messages indicating a problem in *istdops.h*	182
Key Access Function - How to Return the Key (1)	Error messages indicating a problem in *istdops.h*: a local variable or compiler temporary is being used in a return expression	184
Key Access Function - How to Return the Key (2)	Unexpected results when adding an element to a unique key collection	185
Definition of Key-Type Functions	Link step returns error message EDC3013	185
Exception Tracing	Unexpected exception tracing output on standard error	186
Declaration of Template Arguments and Element Functions (1)	Compiler messages (when templates are being processed) indicating that an element type or one of its required element functions is not declared	186
Declaration of Template Arguments and Element Functions (2)	Compilation errors from symbols being defined multiple times	186
Declaration of Template Arguments and Element Functions (3)	Link errors from symbols being defined multiple times	187
Default Constructor	Compiler error messages indicating a problem with constructors	187
Considerations when Linking with Templates	Unresolved external references during linking	188

Cursor Usage

Effect You get unexpected results when using cursors. For example, the `elementAt()` function fails for the given cursor or returns an unexpected element.

Reason You have used an undefined cursor. Cursors become undefined when an element is added to or removed from the collection.

Solution Cursors that become undefined must be rebuilt with an appropriate operation (for example, `locate()`) before they are used again. Rebuilding is especially important for removing all elements with a given property from a collection. Elements cannot be removed by coding a cursor iteration. Use the `removeAll()` function that takes a predicate function as its argument.

For more information about cursors, see “Cursors” on page 98 and “Removing Elements” on page 97.

Element Functions and Key-Type Functions

Effect When compiled, your program causes a compiler error indicating a problem in *istdops.h*. The following are examples of such errors:

Message if key is missing

```
j:\...\ibmclass\istdops.h(166:1) : (E) EDC3013:
  "key" is undefined.
j:\...\ibmclass\istdops.h(160:1) : informational EDC3207:
  The previous message applies to the definition of template
  "IStdKeyOps<Parcel,ToyString>::key(const Parcel&) const".
```

Message if hash is missing

```
j:\...\ibmclass\istdops.h(152:1) : (E) EDC3070:
  Call does not match any argument list for "::hash".
j:\...\ibmclass\istdops.h(146:1) : informational EDC3207:
  The previous message applies to the definition of template
  "IStdHshOps<ToyString>::hash(const ToyString&,unsigned long) const".
```

Message if == is missing

```
j:\...\ibmclass\istdops.h(81:1) : (E) EDC3054:
  The "==" operator is not allowed between "const ToyString" and
  "const ToyString".
j:\...\ibmclass\istdops.h(80:1) : informational EDC3207:
  The previous message applies to the definition of template
  "equal(const ToyString&,const ToyString&)".
```

Message if < is missing

```
j:\...\ibmclass\istdops.h(105:1) : (E) EDC3054:
  The "<" operator is not allowed between "const ToyString"
  and "const ToString".
j:\...\ibmclass\istdops.h(103:1) : informational EDC3206:
  The previous 2 messages apply to the definition of template
  "compare(const ToyString&,const ToyString&)".
```

Reason

Compiler error messages indicating a problem in *istdops.h* are related to the element and key-type functions that you must define for your elements. These functions depend on the collection and implementation variant you are using. The compilation errors listed above occur when the `key()` function, the `hash()` function, `operator==`, or `operator<` are required for your elements, but are defined with the wrong interface or not defined at all. Whether arguments are defined as **`const`** is significant. Compiler messages do not always point directly to the incorrect function. For example, a compare function with non-**`const`** arguments results in the compilation error:

The "<" operator is not allowed between "const ..".

Solution

Verify which element and key-type functions are required for the implementation variant of the collection you are using. You can find this information for each collection in the section pertaining to the collection under the heading “Template Arguments and Required Functions.”

For more information about element and key-type functions, see Chapter 9, “Element Functions and Key-Type Functions” on page 107.

Note that the same problem may be produced if function declarations and definitions are not properly separated between .h files and .cpp files. This situation is described in detail in “Declaration of Template Arguments and Element Functions (1)” on page 186.

Key Access Function - How to Return the Key (1)

Effect

You get a compiler warning similar to:

Message if key is passed by value

```
j:\...\ibmclass\istdops.h(166:1) : warning EDC3285:
  The address of a local variable or compiler temporary is being used
  in a return expression.
j:\...\ibmclass\istdops.h(160:1) : informational EDC3207:
  The previous message applies to the definition of template
  "IStdKeyOps<Word,int>::key(const Word&) const".
```

Reason

Compiler error messages indicating a problem in *istdops.h* are related to the element and key-type functions that you must define for your elements. These functions depend on the collection and implementation variant you are using. Your global-name-space function `key()` returns the key by value instead of by reference. A temporary variable is created for the key within the operator-class function key. The operator class function key returns the key by reference. Returning a reference to a temporary variable causes unpredictable results.

The key function must return a reference and must also take a reference argument. If the key function calls other functions to access the key, it must call those functions with a reference to the object as an argument, and those functions must return a reference to the key.

Solution

Verify that the global name-space function key correctly returns a **`key const&`** instead of **`key`**.

For more information on element and key-type functions, see Chapter 9, “Element Functions and Key-Type Functions” on page 107.

Key Access Function - How to Return the Key (2)

Effect You are adding an element into a unique key collection, such as a key set or a map, and you are sure that the collection does not yet contain an element with the same key. Nevertheless, you get unexpected results: `IKeyAlreadyExistsException`, or the element is not added and the cursor is positioned to a different element.

Reason This problem has the same cause as the problem described in "Key Access Function - How to Return the Key (1)" on page 184. However, you did not get the warning message described above, because you compiled with a lower warning level.

Solution This problem has the same solution as that described in "Key Access Function - How to Return the Key (1)" on page 184.

Definition of Key-Type Functions

Effect You are using a collection class with a key, and you get an error message during the link step indicating a problem in *istdops.h*. The following are examples of such errors:

Message if key() function is undefined

```
istdops.h(176): (E) EDC3013: "key" function is undefined.
```

Reason You are using a collection class that requires the element class to provide a key and you chose to use the method of using a global `key()` function. You are using collection class methods in a .cpp file but the .h file with the same name as the .cpp file does not contain a declaration (prototype) of the global key function.

While compiling the .cpp file, which uses methods of the collection class, the C++ compiler has created or modified a temporary .cpp file in the `tempinc` directory. During the link step, this .cpp file is compiled to resolve references to template code. The error message you encounter refers to this compilation. The .cpp file in the `tempinc` directory contains include directives for the collection class template code. It also contains include directives for a .h file of the same name as the .cpp file that uses the collection class methods. The template code in *istdops.h* requires that the global `key()` function be known at compilation time. The only file that is included at this time is the .h file with the same name as your .cpp file. The problem is that the .cpp file is not included at this time, so a definition or declaration of the global `key()` function in this file is not recognized by the compiler.

Solution You must declare the global `key()` function in the .h file with the same name as the .cpp file that uses the collection class methods. The definition of the global `key()` function should be in the .cpp file. If you are not sure which .h file is meant by the message, look in the .cpp file found in the *tempinc* directory.

Exception Tracing

Effect You get unexpected exception tracing output on standard error, even though the related exception causing the output is caught.

Reason For each exception raised, the trace function `write()` of class `IException::TraceFn` is called and writes information about the raised exception to standard error. This trace function `write()` is called whether the related exception is caught or not.

Solution To suppress the trace output, provide your own `IException::TraceFn::write()` tracing function by subclassing `IException::TraceFn` and register the subclass with `setTraceFunction()`.

For more information about exception tracing, see the *Open Class Library Reference Volumes 2 and 3*.

Declaration of Template Arguments and Element Functions (1)

Effect You get compiler messages when processing templates indicating that an element type or one of its required element functions is not declared.

Reason The element type or element function is defined locally to the .cpp file that contains the template instantiation with the element type as its argument. The prelink phase is executed only by using the header files. Therefore, your declaration local to a .cpp file is not recognized and causes these compilation errors.

Solution Move the corresponding declarations to a separate header file and include the header file from the .cpp file.

Declaration of Template Arguments and Element Functions (2)

Effect You get compilation errors from symbols being defined multiple times.

Reason The template instantiation needs to include the type declarations it received as arguments. Your header files containing type declarations used in template classes may automatically be included several times.

Solution Protect your header files against multiple inclusion by using the following preprocessor macros at the beginning and end of your header files:

```
#ifndef _MYHEADER_H_
#define _MYHEADER_H_ 1

 ⋮
#endif
```

Where *_MYHEADER_H_* is a string, unique to each header file, representing the header file's name.

Declaration of Template Arguments and Element Functions (3)

Effect You get link errors from symbols being defined multiple times.

Reason The template instantiation needs to include the type declarations it received as arguments. Your header files containing type declarations used in template classes might automatically be included several times.

Solution Verify that you did not define functions in the header files that declare types used in templates. If you did, you must move them from the header file into a separate .cpp file or make them inline.

Default Constructor

Effect You get a compiler error similar to the following:

Message for missing default constructor

```
itbseq.h(25:1) : (E) EDC3222:
"IGTabularSequence<ToyString,IStdOps<ToyString> >::Node" needs a
constructor because class member "ivElement" needs a constructor
initializer.
Names namesOfExtinct(animals.numberOfDifferentKeys());
ANIMALS.C(55:57) : informational EDC3207:
The previous message applies to the definition of template
 "ITabularSequence<ToyString>".
```

Reason Compiler error messages indicating a problem with constructors for a collection are typically related to the constructors defined for your element. Here the default constructor for the element is missing.

Solution Define the default constructor for the element class.

For more information about element and key-type functions, see Chapter 9, "Element Functions and Key-Type Functions" on page 107. The element and key-type functions required for each collection are listed for each collection type in sections entitled “Template Arguments and Required Functions.”

Considerations when Linking with Templates

Effect You get unresolved external references during linking that refer to symbols you cannot explain.

Reason A possible reason for unresolved external references during linking is that template code cannot be correctly resolved.

Solution

1. Use *ICC* for linking. *ICC* knows it has to process templates, *LINK386* does not.
2. Use the *-Tdp* option for linking. This tells ICC it is processing C++ code that might have templates, so ICC may have to process these templates.

Part 4. Data Type and Exception Class Library

This part tells you how to use the data type and exception classes. You can use these classes to create and manipulate strings, date and time information, handle exceptions, or define your own classes.

Note: For information on the INotificationEvent, INotifier, IObserver, IObserverList, and IStandardNotifier classes, see *Building VisualAge C++ Parts for Fun and Profit*.

16 Data Types and Exceptions

The Data Types and Exceptions Class Library was developed by IBM, originally as part of the User Interface Class Library on C Set ++ for OS/2. Because these classes did not have the graphical-user-interface orientation of other classes in the User Interface Class Library, the classes were separated from the User Interface Class Library into a library of their own. On some earlier implementations, this class library was known as the “Application Support Class Library.”

Organization of Classes

Figure 23 on page 192 shows the organization of the Data Type and Exception classes that are derived from `IBase` and those that are derived from `IException`. Five other classes do not inherit from any classes and are used to support the derived classes. See Table 5 on page 194 for information on the names of these classes and the classes they support. The purposes of the principal classes are described below. Classes are listed alphabetically.

`IBase` The base class of most of the other classes in the Data Types, Exception, and User Interface classes of IBM Open Class Library. This class provides an output operator and conversion functions for the library, and **`typedef`** synonyms used by other library classes to make programming easier. You do not need to create objects of the `IBase` class; it is described for completeness only.

`IBuffer` Objects of the buffer classes contain the actual character contents of objects of the string classes. All manipulation of string characters is done in the buffer object referenced by the string object. `IBuffer` is the buffer class for single-byte character set objects.

`IDate` This class provides support for date information. You can construct `IDate` objects in a number of ways, and then use `IDate` methods to determine the day of the week, month or year, compare two dates, test a date for certain characteristics, and obtain the names of days or months that are dependent on the national-language locale setting in effect at run time.

`IDBCSBuffer` This class is the buffer class for double-byte character sets. Double-byte character sets are used for handling languages such as Japanese, Chinese, and Korean, which contain more symbols than can be represented by the 256 characters of the single-byte character set.

Class Organization

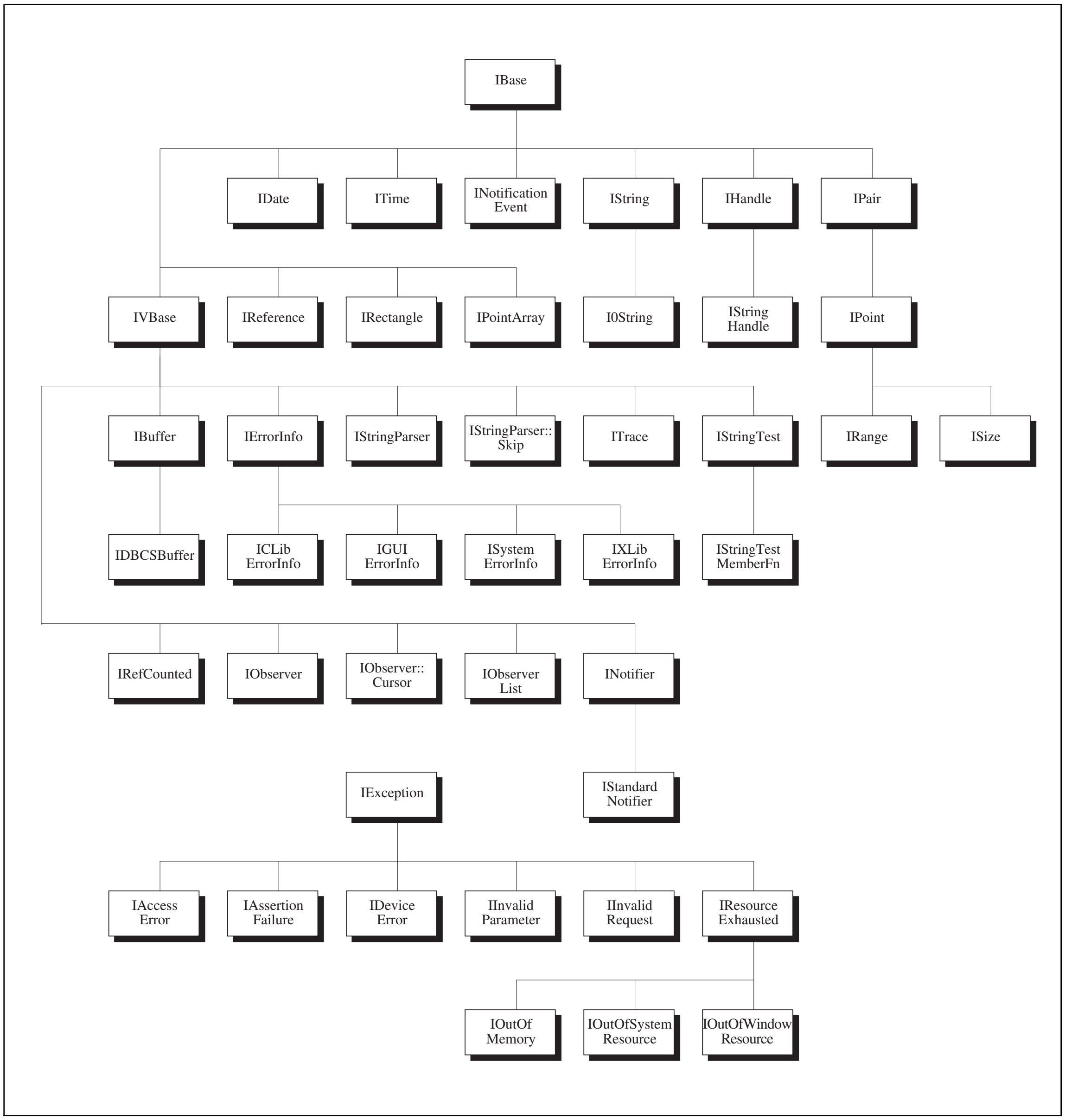


Figure 23. Organization of Data Types and Exceptions Class Library. Some class names have been split into two lines to fit in their boxes.

`IErrorInfo` The `IErrorInfo` class is an abstract base class that defines the interface for its derived classes. These classes retrieve error information and text that is then put into an exception object.

`IException` The `IException` class is the base class from which all exception objects thrown in the library are derived.

`IObserver` This class, along with the `IObserverList`, `INotifier`, `IStandardNotifier`, and `IObserver::Cursor` classes, lets you register observers with class objects so that you can be notified when a change to such an object takes place. For further information on using these classes, see *Building VisualAge C++ Parts for Fun and Profit.*

`IString` This class gives you a greater flexibility in handling strings than traditional C-style character arrays. The `IString` class supports both single- and double-byte character sets. With `IString` objects, you can code string-handling operations much more quickly. For example, you can concatenate two strings simply by using the `+` operator, or compare them using the `==` operator.

`IStringTest` This class is provided so that you can define your own version of the matching function used by `IString` search and compare methods.

`ITime` You can use this class to create time-of-day objects and to compare them, add them together, extract specific information from them, or write them to an output stream.

`ITrace` Objects of the `ITrace` class provide module tracing. Whenever an exception is thrown by the library, trace records are output with information about the exception. You can use environment variables to redirect the trace output to a file.

`IVBase` This class is a virtual base class used to derive other classes such as the buffer classes.

`I0String` This class is identical to the `IString` class, except in its method of indexing strings. In the `IString` class, the first character of a string is at position 1, whereas the same string when stored in an `I0String` object has its first character at position 0. `I0String` is provided for programmers who are used to the C string-handling approach of treating strings as starting at position 0. `IString` and `I0String` objects are easily interchanged, and they support the same set of methods and operators.

One of the most important classes from a programmer's perspective is the `IString` class. This class can make your programming much more productive if you do any amount of string handling. The `IString` class provides a simpler, safer, and more flexible way of handling strings than traditional C-style character arrays and the functions of the `string.h` library. The `IString` class has associated classes that give you even greater flexibility in how you index strings and in how you test for pattern matches in the searching and replacing functions the class provides.

Table 5. Support Classes for Data Type and Exception Classes

Class Name	Supports These Classes
IStringEnum	IString I0String IBuffer IDBCSBuffer
IMessageText	IBase
IException::TraceFn	IException
IExceptionLocation	IException

IBase Class

The IBase class provides:

- An output operator for the library
- Conversion functions for the library
- Handling of the message text file
- Types for the library
- Synonyms

You do not need to create objects of the IBase class. This class is introduced at the root of the class hierarchy for the following reasons:

- To define the local type Boolean and the enumeration values true and false. This definition enables these identifiers to be referenced without their scope qualifier IBase:: within declarations and member function definitions of classes derived from IBase.
- To provide basic functions applicable to many of the classes in IBM Open Class Library. These functions are asString(), asDebugInfo(), and operator<<(ostream&). Note that asString() and asDebugInfo() do not work correctly if they are invoked through a pointer or reference to an IBase object, because the functions are not virtual. IVBase redeclares these as virtual functions. This means that, if you invoke these functions against an IVBase* or IVBase& object, the implementation for the actual class of the pointed-to or referenced object is invoked.

IVBase Class

The IVBase class:

- Ensures generic behavior for library classes that have virtual functions

- Allows derived classes to access the type and value names of the IBase class

All functions in the IVBase class should be overridden in derived classes because the IVBase class does not have access to any useful information about objects of its derived classes.

String and Buffer Classes

You can store and manage strings using the string and buffer classes. There are two type of string classes, two types of buffer classes, and two support classes. The two string classes, IString and I0String, are the main classes. The buffer and support classes are used to implement the string classes.

The buffer classes, IBuffer and IDBCSBuffer, contain the actual contents of the string objects. The IDBCSBuffer class supports characters of the double-byte character set (DBCS). If you are using the string classes, DBCS support is automatic and transparent.

IBuffer and IDBCSBuffer are purely internal classes used in the implementations of IString and I0String. They are only used in protected sections of the IString class. They are described in this guide because you may want to understand them if you are deriving classes from IString.

The support classes, IStringEnum and IStringTest, provide data types and testing functions that are used in the string and buffer classes.

DBCS and National Language Support

The library provides double-byte character set (DBCS) support and national language support (NLS). You can use one source file for your application code and provide DBCS and NLS support by using separate resource files for the languages you support. The benefits of this organization include the following:

- The application is easy to maintain, because a single version of the application is used. This reduces the cost of maintaining your code.
- The application is easy to upgrade because only the source code is upgraded and then linked to the separate language files for different languages. This reduces the time and cost of upgrading your code because different language versions can be generated at the same time.

Because message strings are defined in resource files, they can be translated easily to your local language without changes to the source code.

You should note the following when creating a DBCS-enabled application:

DBCS and National Language Support

- String manipulation is DBCS-enabled. The string classes support mixed strings that contain both SBCS and DBCS characters. Use the string testing functions to determine if a character is single byte or double byte.
- The `IDBCSBuffer` class ensures that the search functions do not match the second or any subsequent bytes of a DBCS character and that the bytes of a DBCS character will not be split.

17

String Classes

The string classes define a data type for strings and provide member functions that let you perform a variety of data manipulation and management activities. They provide capabilities far beyond those available with standard C strings and the `string.h` library functions.

The string classes have the following capabilities:

- String buffers are handled automatically.
- Strings can contain both SBCS and DBCS characters.
- Strings can be indexed by character or by word.
- Strings can contain null characters. (There are no restrictions on the contents of a string object.)

Member functions of the string classes allow you to:

- Use strings in input and output
- Access information about strings
- Compare strings
- Test the characteristics of strings
- Search for characters or words within a string
- Manipulate and edit strings
- Convert strings to and from numeric types
- Format strings by adding or removing white space

Introduction to the String Classes

There are two string classes: `IString` and `I0String`. They are identical except for the method each uses to index its characters. The characters of an `IString` object are indexed beginning at 1. `I0String` characters are indexed beginning at 0. 📖 See “Indexing of Strings” on page 198 for more information on the indexing of the string classes. The string class you should use depends on which indexing scheme you prefer or find easier to implement.

Objects of `IString` and objects of `I0String` can be freely intermixed in a program. Objects of one class can be assigned objects of the other. Arguments that require an object of one will accept objects of the other. You will only notice a difference between an `IString` and an `I0String` when you are using functions that use or return a character index value.

In this chapter, only the `IString` class is presented. However, for every function of the `IString` class, there is a corresponding and identically named function of the

IOString class. The IOString version of each function accepts the same arguments and has the same return type as the IString version, except that all parameters of type IString become IOString. Any other differences between the IString and IOString versions of the function are noted in the function descriptions in the *Open Class Library Reference*.

String Buffers

When you create an object of a string class, the actual characters that make up the string are not stored in the string object. Instead, the characters are stored in an object of a buffer class.

The use of a buffer object is transparent to you when using the string classes. A correctly sized buffer is automatically created when you create a string object. The buffer is destroyed when a string object is destroyed. When you manipulate or edit a string, you are actually manipulating and editing the buffer object that contains the characters of the string.

Double-Byte Character Set Support

Objects of the IString class and the IOString class can contain a mixture of single-byte characters and double-byte characters. All member functions allow for the mixture. The searching functions will not match a single-byte character with the second or subsequent byte of a double-byte character. Functions that return substrings will never separate the bytes of a double-byte character.

Although the double-byte characters are supported, you must be careful not to alter the contents of a string in a way that would corrupt the data. For example, the statement:

```
IString[n]='x';
```

would be an error if the nth byte of the IString was part of a double-byte character.

Indexing of Strings

Objects of the string classes are arrays of characters. There are two types of indexes used with the arrays. The first is a character index: each character is numbered from left to right starting at the number 1 in the IString class and the number 0 in the IOString class. Therefore in the IString “The dog is brown,” the letter “i” has an index value of 9. In the IOString “The dog is brown,” the letter “i” has an index value of 8.

The second type of index is the word index. In the word index, each white-space-delimited word is numbered from left to right starting at the number 1. The word index is the same for IString objects and IOString objects. Therefore in the IString “The dog is brown,” the word “is” has an index value of 3. In the I0String “The dog is brown,” the word “is” also has an index value of 3.

The only difference between objects of the IString class and objects of the I0String class is the starting value for the character index.

What You Can Do with Strings

This section describes the wide range of string handling capabilities provided by the IString class. If you have a particular task you want to learn about from the list below, you can look that task up now and find references to appropriate IString functions. If you want an overview of all the capabilities of the IString class, read the entire section. The tasks are:

- Creating and copying strings
- Doing string input and output
- Concatenating strings
- Finding words or substrings within strings
- Replacing, inserting, and deleting substrings
- Determining string lengths and word counts
- Extending strings
- Converting between strings and numeric data
- Converting between strings and different base notations
- Testing the characteristics of strings
- Formatting strings

Many of the IString operators and functions are overloaded to support both IStrings and arrays of characters as return types and arguments. For example, the comparison operators (==, >, <, >=, <=, !=) all support either two IString operands or one IString and one array of characters operand. The array of characters operand can be on either side of the comparison operator. See the descriptions of individual member functions in the *Open Class Library Reference* to determine what combinations of IString and array of characters are supported for a given function or operator.

Creating and Copying Strings

You can create IStrings using constructors, and you can copy IStrings using copy constructors, assignment operators, and substring functions.

IString Constructors

You can use IString constructors that construct null strings, that accept a numeric argument and convert it into a string of numeric characters, or that translate one or more characters into an IString. You can also create a single string out of up to three separate buffers, whose contents are concatenated into the created IString object. The following example shows some of the above ways of creating IString objects:

```
#include <istring.hpp>
#include <iostream.h>
void main() {
   IString Number1(123);    // --> Number1   ="123"
   IString Number2(123.12); // --> Number2   ="123.12"
   IString Character('a');  // --> Character ="a"
   IString String1("a");    // --> String1   ="a"
   IString String2("and");  // --> String2   ="and"
   IString String3("a\0d"); // --> String3   ="a"
   }
```

Note that the last string (String3) is initialized with only the first byte of quoted text. The null character in the **char*** constructor argument is interpreted by the compiler as a terminating null. However, the IString class does support null bytes within strings. To construct String3 as the example intended, you could write:

```
//...
IString String3("and");
String3[2]='\0';
```

If this string is later copied to another string, the null character and following characters are also copied:

```
IString String4=String3;
String4[2]='N';          // --> String4   ="aNd"
```

Copying IStrings

The IString assignment operator and copy constructor both copy one string to another string. One of the strings can be an array of characters, or both may be IString objects. The IString assignment operator and copy constructor offer the following advantages over the strcpy and strdup functions provided by the C string.h library:

- When an IString object is copied, a new copy of the string is not made. Instead, the two strings point to the same buffer location. The object is only copied if one of the strings is changed. This means that, for strings that are copied but where neither the source string nor the copy is subsequently changed, performance is improved by the amount of time it would have taken to make the new copy.
- The notation is simple and intuitive. To copy String1 into String2, you simply code String2=String1. With strings defined using the traditional **char*** method, such an assignment merely copies a pointer to the original string. With IString objects, the assignment copies each byte of the string into the new string.
- You do not have to determine the length of the source string and allocate sufficient storage to store it in the target string before the assignment. IString takes care of allocating the storage for you, whether the target string is being constructed within the assignment or has already been constructed. This reduces

the risk of memory violations. In the following example, String2 is constructed and initialized, and then copied to (its original contents are overwritten), while String3 is copy-constructed to contain a copy of String1. Notice that String2's length is extended by the assignment operation.

```
IString String1="A longer string than String2";
IString String2="A short string";
IString String3=String1;    // initialized to String1
String2=String1;            // extended to fit String1
```

- The string being copied can contain null characters anywhere within it, and the entire string will be copied.
- If you accidentally create an array of characters without the terminating null, the strcpy function may continue copying past the storage allocated for the string. This can cause storage violations, or, at the least, it can corrupt the data in the target string. The length of IString objects is not determined by a terminating null, so storage violations and corrupt target strings are less likely.

Creating Substrings of Strings

You can use the subString function to return a new IString object containing a portion of another IString. This function lets you create an IString containing the leftmost characters, rightmost characters, or characters in the string's middle. The following example shows calls to subString that create substrings with leftmost, rightmost, or middle characters:

```
// Using the subString method of IString

#include <iostream.h>
#include <istring.hpp>

void main() {
   IString All("This is the entire string.");

   // Left -> subString(1, length)
   IString Left=All.subString(1,5);

   // Middle -> (startpos, length)
   IString Middle=All.subString(6,14);

   // Right -> (string length - (substring length - 1) )
   IString Right=All.subString(All.length()-6);

   cout << "<" << All << ">\n"
        << "<" << Left << ">\n"
        << "<" << Middle << ">\n"
        << "<" << Right << ">" << endl;
   }
```

This program produces the following output:

```
<This is the entire string.>
<This >
<is the entire >
<string.>
```

Doing String Input and Output

The IString class overloads the input and output operators of the I/O Stream Class Library so that you can extract IString objects from streams and insert IString objects into them. The input operator reads characters from the input stream until a white-space character or EOF is encountered. The IString class also defines a member function to read a single line from an input stream. The following example shows uses of the input and output operators for IString and the lineFrom function:

```
//Using the IString I/O operators and the lineFrom function

#include <istring.hpp>
#include <iostream.h>

void main() {
    IString Str1, Str2, Str3;
    Str1="Enter some text:";
    char test[80];

    // Write prompt
    cout << Str1;
    // Get input
    cin >> Str2;
    // This only reads in one word of text, so we should
    // check to see if this was the only word on the line:
    if (cin.peek()!='\n') {
       // there's more text on this line so ignore it
       cin.ignore(1000,'\n');
       }
    // Change prompt
    Str1.insert("more ",Str1.indexOf(" text:"));
    // Write prompt again
    cout << Str1;
    // Get line of input
    Str3=IString::lineFrom(cin,'\n');
    // Write output
    cout << "First word of first input: " << Str2 << '\n'
         << "Full text of second input: " << Str3 << endl;
    }
```

This example produces the output shown below in regular type, given the input shown in bold:

```
Enter some text:Here is my first string
Enter some more text:Here is my second string
First word of first input: Here
Full text of second input: Here is my second string
```

Note that, although null characters are allowed within an IString object, a null character in an input string is treated as the end of the input, and a null character in an IString being written to an output stream ends the output of that IString.

Concatenating Strings

The IString class defines an addition operator (+) to allow you to concatenate two words together. An addition assignment operator (+=) lets you assign the result of the concatenation to the left operand. The copy() member function lets you create an

IString consisting of multiple copies of itself or of another string. The following example shows ways of concatenating text onto the start or end of an IString:

```
// Concatenating strings

#include <iostream.h>
#include <istring.hpp>

void main() {
   IString Str1="Let ";
   IString Str2="us ";
   IString Str3="concatenate ";
   IString Str4="repeatedly ";

   IString Str5=Str1+Str2;  // Add Str1 and Str2 and store in Str5;
   Str5+=Str3;              // Add Str3 to Str5
   Str4.copy(3);            // Copy Str4 several times onto itself
   Str5+=Str4;              // Add Str4 to Str5
   cout << Str5 << endl;    // Write String 5
   }
```

This program produces the following output:

```
Let us concatenate repeatedly repeatedly repeatedly
```

Finding Words or Substrings within Strings

A wide range of functions are available to let you find words, substrings, patterns, or individual characters within a string. You can even do wildcard searches: for example, you can search through a string to find a substring that begins with the letters "Ar" followed by one or more characters, followed by the letters "rk".

The following example shows a number of the searching functions available for IString objects. Comments describe the type of search operation being carried out.

```
// Searching for substrings

#include <iostream.h>
#include <istring.hpp>
void main() {
   IString Str1="This string contains some sample text in English.";
   IString Str2=Str1.subString(27); // positions 27 and following:
                                    // "sample text in English."
   cout << "The string under consideration is:\n\n"
        << Str1 << "\n\n";

// 1. Count the number of occurrences of a substring within the string

   cout << "The substring \"in\" occurs "
        << Str1.occurrencesOf("in")
        << " times in the string.\n";

// 2. Find the first occurrence of a substring:
//    (Note that the substring can be a char, char*, or IString value)

   cout << "The letter 'x' first occurs at position "
        << Str1.indexOf('x') << ".\n";

// 3. Find the first occurrence of any letter of those specified:

   cout << "One of the letters q, r, or s first appears at position "
        << Str1.indexOfAnyOf("qrs") << ".\n";
// 4. Find the first occurrence of any letter other than those specified:

   cout << "The first letter that is not in \"Think\" "
        << "appears at position "
        << Str1.indexOfAnyBut("Think") << ".\n";

// 5. Find the index of a word

   cout << "The third word starts at position "
        << Str1.indexOfWord(3) << ".\n";

// 6. Find a match to a phrase, and return the position of the
//    first matching word

   cout << "The phrase \"" << Str2 << "\" starts at word number "
        << Str1.wordIndexOfPhrase(Str2) << " of the string.\n";

// 7. Do a wildcard search to see if the string starts with "Th",
//    contains "co", and ends with "sh."

   cout << "Does the string match the wildcard search string "
        << "\"Th*co*sh.\"?\n";
   if (Str1.isLike("Th*co*sh.")) cout << "Yes.";
   else cout << "No.";

   cout << endl;
   }
```

This program produces the following output:

```
The string under consideration is:

This string contains some sample text in English.

The substring "in" occurs 3 times in the string.
The letter 'x' first occurs at position 36.
One of the letters q, r, or s first appears at position 4.
The first letter that is not in "Think" appears at position 4.
The third word starts at position 13.
The phrase "sample text in English." starts at word number 5 of the string.
Does the string match the wildcard search string "Th*co*sh."?
Yes.
```

Replacing, Inserting, and Deleting Substrings

The ability to manipulate the contents of an `IString` is one of the greatest advantages of the `IString` class over the traditional method of using `string.h` functions to manipulate arrays of characters. Consider, for example, a function that perform the following changes on a string. Issues that you need to address when using arrays of characters, but that are handled for you by the `IString` class, are shown in parentheses:

1. Replace all occurrences of `Blue` with `Yellow` (string must be expanded by two characters for each replacement, and text after the replacement must be shifted out).

2. Replace all occurrences of `Orange` with `Pink` (string must be shortened by two characters for each replacement).

3. Delete the sixth word of the string. (How are words delimited? By spaces? Carriage returns? Tab characters? What about multiple adjacent whitespace characters?)
4. Insert the word `Dark` as the fourth word or at the end of the string if the string has fewer than three words. (String must be extended. How are words delimited? Do you add a space before or after the word?).

You can easily handle the above requirements using `IString` member functions. The sample function `fixString()` below implements the requirements. Numbered comments correspond to the numbers of the requirements:

```
// Inserting, deleting and replacing substrings

#include <iostream.h>
#include <istring.hpp>

void fixString(IString&);

void main() {
   IString Str1="Light Blue and Green are nice colors. ";
   Str1+="But so are Red and Orange.";
   cout << Str1 << endl;
   fixString(Str1);
   cout << Str1 << endl;
   }

void fixString(IString &myString) {
   myString.change("Blue", "Yellow");   // 1. Change Blue to Yellow
   myString.change("Orange", "Pink");   // 2. Change Orange to Pink
   myString.removeWords(6,1);           // 3. Remove words, starting at word 6,
                                        //    for a total of 1 word.
   int Word4=myString.indexOfWord(4);
   if (Word4>0)                         // 4. Insert "Dark" as fourth word
      myString.insert("Dark ",Word4-1); //    or at end of string if string
   else                                 //    has fewer than 4 words.  The
      myString+=" Dark";                //    insertion occurs 1 byte before
   }                                    //    word 4 (otherwise it inserts
                                        //    in the middle of word 4).
```

This program produces the following output:

```
Light Blue and Green are nice colors. But so are Red and Orange.
Light Yellow and Dark Green are colors. But so are Red and Pink.
```

Determining String Lengths and Word Counts

You can determine not only the length of a string, but the number of words within the string, or the length of a particular word in the string. The length of a string is not affected by any null characters you insert in the middle of the string. (The `strlen` function of `string.h` treats any null character in an array of characters as a terminating null.)

The following descriptions assume that `ThisString` contains the text “This string has five words.”

The `length` and `size` functions both return the length of an `IString`. For example, `ThisString.size()` returns the value 26, as does `ThisString.length()`.

To determine the number of words in a string, use the `numWords` member function. For example, `ThisString.numWords()` returns the value 5.

To determine the length of a particular word, use the `lengthOfWord` member function. For example, `ThisString.lengthOfWord(3)` returns the value 3.

Extending Strings

With arrays of characters, unless you allocate more storage than originally required for a string, you can only extend a string by allocating a new chunk of storage, moving the existing string into the new area, and extending it there.

`IString` objects are automatically extended for you whenever an `IString` operator or function requires the extension. This lets you spend more time coding useful function, and less time trying to track down the source of memory violations or data corruption. You can even use the subscript operator to assign a value to a position beyond the end of the string. The following example, by indexing past the end of `ShortString`, causes the string to be padded with blanks up to position 119, and the letter “a” is added at position 120:

```
IString ShortString="A short string";
ShortString[120]='a';
```

The `+` and `+=` operators, the assignment operator, and all member functions that change the contents of a string automatically allocate additional storage for the string if that storage is required. This can drastically reduce the amount of string-handling code you need to write.

Converting between Strings and Numeric Data

The `IString` class provides a number of `as...` functions that convert from `IString` objects to numeric types. You can also convert from numeric types to `IString` objects by using the versions of the `IString` constructor that take numeric values as arguments. The following example shows various `IString` functions that convert between strings and numbers:

```
// Conversion between IString and numeric values

#include <iostream.h>
#include <istring.hpp>
void main() {
IString NumStr=1.4512356919E1;   // Initialized with a float value
int Integer=NumStr.asInt();      // Convert to integer value
float Float=NumStr.asDouble();   // C++ conversion rules allow asDouble's
                                 // result to be converted to float
double Double=NumStr.asDouble(); // Convert to double value
NumStr=688;                      // Assign another integer value
cout.precision(20);             // Set precision of cout stream
cout <<   "Integer: " << Integer << "\nFloat:   " << Float
     << "\nDouble:  " << Double  << "\nString:  " << NumStr << endl;
}
```

This program produces the following output:

```
Integer: 14
Float:   14.512356758117676
Double:  14.512356919
String:  688
```

You can also change the base notation of `IString` objects containing integer numbers, by using the `d2...` functions, which convert from decimal to binary, hexadecimal, or character representations. Conversion functions are described in the next section.

Converting between Strings and Different Base Notations

You can use the format conversion functions to change the way the data in a string is represented. These functions are overloaded so that each function has two versions. The nonstatic version replaces the value of the string with the converted value. The static version preserves the original string and returns a new string object containing the converted value. For example:

```
aString.c2b();                        // Changes value of aString
IString binaryDigits = c2b( aString ); // Preserves value of aString
```

The conversion functions check the format of the source string to make sure it is compatible with the source format implied by the function name. For example, if you use the `b2d` function to convert a string from binary to decimal, the function first checks that the string contains only the digits ‘0’ and ‘1’. If it contains any characters other than those allowed by the source type, the format conversion functions always return 0.

The following example shows the use of the conversion functions. If you examine both the example and the output provided below, you can see how to use the functions.

```
// IString conversion functions

#include <istring.hpp>
#include <iostream.h>
enum Bases {Bin, Dec, Hex, Char};
IString Base[4]={"binary", "decimal", "hex", "character"};
IString NumStr;

void Show(int From, int To, IString& Result) {
   cout << NumStr << " in " << Base[From] << " is "
        << Result << " in " << Base[To] << '.' << endl;
   }
```

```
void main() {
   IString NewStr;
   NumStr="122";
      NewStr=IString::d2b(NumStr); Show(Dec,Bin,NewStr);
      NewStr=IString::d2x(NumStr); Show(Dec,Hex,NewStr);
      NewStr=IString::d2c(NumStr); Show(Dec,Char,NewStr);
   NumStr="Hat";
      NewStr=IString::c2b(NumStr); Show(Char,Bin,NewStr);
      NewStr=IString::c2d(NumStr); Show(Char,Dec,NewStr);
      NewStr=IString::c2x(NumStr); Show(Char,Hex,NewStr);
   NumStr="5F";
      NewStr=IString::x2b(NumStr); Show(Hex,Bin,NewStr);
      NewStr=IString::x2d(NumStr); Show(Hex,Dec,NewStr);
      NewStr=IString::x2c(NumStr); Show(Hex,Char,NewStr);
   NumStr="0110100001101001";
      NewStr=IString::b2d(NumStr); Show(Bin,Dec,NewStr);
      NewStr=IString::b2x(NumStr); Show(Bin,Hex,NewStr);
      NewStr=IString::b2c(NumStr); Show(Bin,Char,NewStr);
   }
```

The output from this program resembles the following. Depending on the code page and character set (ASCII or EBCDIC) of the system you are running the program on, the values may vary.

```
122 in decimal is 01111010 in binary.
122 in decimal is 7A in hex.
122 in decimal is z in character.
Hat in character is 010010000110000101110100 in binary.
Hat in character is 4743540 in decimal.
Hat in character is 486174 in hex.
5F in hex is 01011111 in binary.
5F in hex is 95 in decimal.
5F in hex is _ in character.
0110100001101001 in binary is 26729 in decimal.
0110100001101001 in binary is 6869 in hex.
0110100001101001 in binary is hi in character.
```

Testing the Characteristics of Strings

The IString class lets you test your strings to determine characteristics such as the following:

- Whether they represent valid hexadecimal, decimal, or binary values
- Whether they contain only letters, letters and numbers, uppercase letters, lowercase letters, or punctuation characters
- Whether they contain all SBCS or DBCS characters

This list covers only a few of the testing functions provided by IString.

The testing functions return a value of type Boolean or IBoolean, indicating either True or False for the tested characteristic. For example, the function isBinaryDigits() returns False for the IString value “1101121101.”

The testing functions all have names beginning with is..., because they ask a question, such as “is the IString made up only of binary digits?” For a complete list of the testing functions, see the *Open Class Library Reference*. The following example shows how you can use a subset of these functions:

```
// Evaluating strings using the IString is... methods

#include <istring.hpp>
#include <iostream.h>

void evaluate(IString& StringToTest) {
   if (StringToTest.isPrintable())
      cout << "Evaluating the string " << StringToTest << ":" << endl;
   else
      cout << "Evaluating an unprintable string:" << endl;
   if (StringToTest.isDigits())
      cout << "   Contains only digits 0-9." << endl;
   if (StringToTest.isAlphabetic())
      cout << "   Contains only alphabetic characters." << endl;
   if (StringToTest.isAlphanumeric())
      cout << "   Contains only alphabetic and numeric characters." << endl;
   if (StringToTest.isBinaryDigits())
      cout << "   Contains only zeros and ones." << endl;
   if (StringToTest.isHexDigits())
      cout << "   Contains only hex digits 0-9, a-f, A-F." << endl;
   if (StringToTest.isControl())
      cout << "   Contains only ASCII values 00-1F, 7F." << endl;
   if (StringToTest.isLowerCase())
      cout << "   Contains only lowercase letters a-z." << endl;
   if (StringToTest.isUpperCase())
      cout << "   Contains only uppercase letters a-z." << endl;
   if (StringToTest.isSBCS())
      cout << "   Contains only SBCS characters." << endl;
   }

void main() {
   IString Str[6];
   Str[0]="12345";              // numeric, hexadecimal
   Str[1]="abcde";              // alphabetic, hexadecimal
   Str[2]="10101";              // numeric, binary
   Str[3]="abCde";              // alphabetic, hexadecimal
   Str[4]="xyz12";              // alphanumeric, lowercase
   Str[5]="\x04\x06\x11\x12";   // control, unprintable

   for (int i=1;i<6;i++) evaluate(Str[i]);
   }
```

The output from this program resembles the following. Depending on the code page and character set (ASCII or EBCDIC) of the system you are running the program on, the results may vary.

```
Evaluating the string abcde:
   Contains only alphabetic characters.
   Contains only alphabetic and numeric characters.
   Contains only hex digits 0-9, a-f, A-F.
   Contains only lowercase letters a-z.
   Contains only SBCS characters.
Evaluating the string 10101:
   Contains only digits 0-9.
   Contains only alphabetic and numeric characters.
   Contains only zeros and ones.
   Contains only hex digits 0-9, a-f, A-F.
   Contains only SBCS characters.
Evaluating the string abCde:
   Contains only alphabetic characters.
   Contains only alphabetic and numeric characters.
   Contains only hex digits 0-9, a-f, A-F.
   Contains only SBCS characters.
```

```
Evaluating the string xyz12:
   Contains only alphabetic and numeric characters.
   Contains only SBCS characters.
Evaluating an unprintable string:
   Contains only ASCII values 00-1F, 7F.
   Contains only SBCS characters.
```

Formatting Strings

You can insert padding (white space) into strings so that each string in a group of strings has the same length. The `center`, `leftJustify`, and `rightJustify` functions all do this; their names indicate where they place the existing string relative to the added white space. You provide the final desired length of the string, and the function adds the correct amount of white space (or removes characters if the string is longer than the final length you specify). For example:

```
// Padding IStrings

#include <istring.hpp>
#include <iostream.h>

void main() {
IString s1="Short", s2="Not so short",
        s3="Too long to fit in the desired field length";
s1.rightJustify(20);
s2.center(20);
s3.leftJustify(20);
cout << s1 << '\n' << s2 << '\n' << s3 << endl;
}
```

This program produces the following output:

```
               Short
    Not so short
Too long to fit in t
```

If a string is too wide, you can strip leading or trailing blanks using the `strip...` functions:

```
// Using the strip... functions of IString

#include <istring.hpp>
#include <iostream.h>

void main() {
   IString s1,  s2,  s3,  Long="      Lots of space here     ";
   s1 = s2 = s3 = Long;
   s1.stripLeading();
   s2.stripTrailing();
   s3.strip();
   cout << ">" << Long << "<\n"
        << ">" << s1   << "<\n"
        << ">" << s2   << "<\n"
        << ">" << s3   << "<" << endl;
}
```

This program produces the following output:

```
>      Lots of space here     <
>Lots of space here     <
>      Lots of space here<
>Lots of space here<
```

You can also change the case of an IString to all uppercase or all lowercase:

```
// Changing the case of IStrings

#include <iostream.h>
#include <istring.hpp>

void main() {
   IString Upper="MANY of THESE are UPPERCASE CHARACTERS";
   IString Lower="Many of these ARE lowercase characters";
   Upper.change("MANY","NONE").lowerCase();
   Lower.change("Many","None").upperCase();
   cout << Upper << '\n' << Lower << endl;
   }
```

This program produces the following output:

```
none of these are uppercase characters
NONE OF THESE ARE LOWERCASE CHARACTERS
```

Other IString Capabilities

This section has described only a portion of the functionality of the IString class. Many functions described here are overloaded to provide a wider range of functionality, and many of the functions of the IString class were not described here. See the *Open Class Library Reference* for complete descriptions of all the public IString functions.

IStringTest Class

The IStringTest class lets you define the matching function used in the searching and testing functions of the string and buffer classes. When a search string is passed to a searching or testing function, the search string and the string object are compared on a character-by-character basis. The characters are considered to match if they are identical. The IStringTest class allows you to define when characters are considered to match.

For example, you can implement a string test that locates a given occurrence of a particular character in a string:

```
// Using the IStringTest class

#include <istring.hpp>
#include <iostream.h>

class Nth : public IStringTest {
   char key;            // Specifies the character to look for
   unsigned count;      // Specifies which occurrence to find
   public:
      //
      // Construct an Nth object as follows:
      // 1. Create an IStringTest instance whose function type is user,
      //    with a null character to start;
      // 2. Initialize the count to n
      // 3. Initialize the key to c
      //
      Nth(char c, unsigned n)
      : IStringTest(user,0), count(n), key(c) { }
      //
      // test function: accepts an int (the character to look for)
      // checks if the character matches the key
      // if so, decrements count
      // eventually, count will equal zero if enough matches are found,
      // so "return !count" will return true (-1)
      // otherwise, "return !count" will return a value other than -1

      virtual Boolean test (int c) const
         {
         if (c == key)          // if it matches,
         ((Nth*)this)->count--; // decrement count
         return !count;         // return complement of count
                                // will be true (-1) if count==0
         }
};

void main() {
IString text="this is a test string";
cout << "The fourth appearance of the letter t in the string:\n"
     << text << '\n' << "is at position "
     << text.indexOf(Nth('t',4)) << endl;
}
```

This program produces the following output:

```
The fourth appearance of the letter t in the string:
this is a test string
is at position 17
```

A derived template class, IStringTestMemberFn, is provided to support the use of the IStringTest class with any function that accepts its objects as an argument.

A constructor for IStringTest accepts a pointer to a C function. The C function must accept an integer as an argument and return a Boolean. Such functions can be used anywhere an IStringTest can be used. Note that this is the type of the standard C library functions that check the type of C characters, for example, **isalpha()** and **isupper()**.

18

Exception and Trace Classes

This chapter outlines some of the ways that you can use the exception and trace classes. The exception classes are a set of classes that allow you to catch exceptions based on their type. The trace class `ITrace` allows you to conveniently put trace statements in your programs.

Introduction to the Exception Classes

There are three primary ways to use the exception classes:

1. Certain functions in IBM class libraries throw exceptions that are objects of the exception classes. If you are familiar with the characteristics of the exception classes, you can take advantage of the exception classes to make your code that uses the IBM class libraries more robust.
2. You can both throw and catch objects of the exception classes in your own code. The exception classes provide a convenient way to package information about an exception.
3. You can derive your own classes from the exception classes.

Characteristics of the Exception Classes

The exception classes have the following characteristics:

- A stack of exception message text strings. These strings allow you to describe the exception in detail.
- An error ID that lets you uniquely identify what error caused the exception.
- A severity code that lets you determine whether the exception can be recovered from or not.
- Information about where the exception was thrown.

The exception classes' member functions allow you to:

- Add information about where the exception was thrown
- Add text to the description of the exception
- Get the error ID of the exception
- Determine if the exception is recoverable
- Log the exception data
- Set the error ID of the exception
- Set the severity of the exception
- Set a trace function

Derivation of the Exception Classes

The exception classes consist of a base class IException and a set of derived classes:

- IAccessError
- IAssertionFailure
- IDeviceError
- IInvalidParameter
- IInvalidRequest
- IResourceExhausted

In addition, IResourceExhausted has the following derived classes:

- IOutOfMemory
- IOutOfSystemResource
- IOutOfWindowResource

Because all these classes are derived from the IException class, a single catch statement can catch all of the exceptions that are objects of the exception classes. The following catch statement, for example, will catch any exception that is an object of one of the exception classes:

```
catch(IException &ie){
     // ...
     // code for all exception class exceptions
}
```

On the other hand, if you wanted to deal with each kind of exception separately, you could have catch statements that looked like this:

```
catch(IAccessError &ia){
     // ...
     // code for IAccessError exceptions
}
catch(IAssertionFailure &if){
     // ...
     // code for IAssertionFailure exceptions
}
// ...
```

Situations in Which the Exception Classes Are Used

The following table lists the exception classes and the situations in which they are typically thrown:

Exception Class	Thrown When ...
IAccessError	A logical error occurs, such as "resource not found"
IAssertionFailure	The expression in an IASSERT macro evaluates to false
IDeviceError	A hardware-related error occurs
IInvalidParameter	An invalid parameter is passed
IInvalidRequest	An object is in the wrong state for a function
IResourceExhausted	A resource is exhausted or currently unavailable
IOutOfMemory	Memory is exhausted

Catching Exceptions Thrown by Class Library Functions

Under certain circumstances, member functions of IBM Open Class will throw exceptions that are objects of the exception classes. You can take advantage of this fact to make your code that uses these classes more robust.

An Example of the new Operator Throwing an Exception

For example, suppose that you use the new operator to create a huge array of integer pointers. If there is not enough memory available to satisfy a particular request for memory, the new operator throws an IOutOfMemory exception.

In the following piece of code, a single invocation of the new operator exhausts all of the memory that is available for allocation. In this code, the catch statement specifies the base class IException rather than IOutOfMemory. If you know that a member function may throw an exception class object, but you do not know its exact type, you can specify a catch statement like this one to catch all of the possible exception class exceptions.

```
//   The new operator throwing an exception

   #include <iostream.h>
   #include <iexcept.hpp>
   #include <istring.hpp>

   #define TOOBIG 1000000000

   void main() {
      int i;
      try {
         int* istr = new int[TOOBIG];
      }
      catch(IException &ie)
      {
        cout << "Type of exception is: " << ie.name() << endl;
        cout << "Location of exception is: "
             << ie.locationAtIndex(0)->fileName() << endl;
```

```
     if (ie.isRecoverable())
        cout << "Exception is recoverable" << endl;
     else
        cout << "Exception is unrecoverable" << endl;
   }
}
```

Assuming that the constant TOOBIG is large enough to exhaust all of the memory available for allocation, this code produces the following output:

```
Type of exception is: IOutOfMemory
Location of exception is: ibase.C
Exception is unrecoverable
```

An Example of the Subscript Operator Throwing an Exception

The subscript operator of the IString class can throw exceptions that are objects of the exception classes. If you use the subscript operator on an IString object that is declared **const**, the operator will throw an InvalidRequest exception if the index is out of the bounds of the IString object.

In the following piece of code, an IString object is declared **const**, and then the subscript operator is used with an index beyond the size of the object.

```
// Example that causes a subscript out of bounds exception

#include <iostream.h>
#include <iexcept.hpp>
#include <istring.hpp>

void main() {
   try {
      const IString ConstStr = "OFF";
      cout << ConstStr[4] << endl;
   }
   catch(IException &ie)
   {
     cout << "Type of exception is: " << ie.name() << endl;
     cout << "Location of exception is: "
          << ie.locationAtIndex(0)->fileName() << endl;
     if (ie.isRecoverable())
        cout << "Exception is recoverable" << endl;
     else
        cout << "Exception is unrecoverable" << endl;
   }
}
```

Because the index is beyond the size of the IString object, the subscript operator throws an exception. When this code is run, the following output is produced:

```
Type of exception is: IInvalidRequest
Location of exception is: istring5.C
Exception is recoverable
```

Member functions in the Collections and User Interface class libraries also throw exceptions that are objects of the exception classes. If you call such functions within try blocks followed by a catch statement for IException exceptions, you can:

- Make your code more robust by detecting and dealing with exceptions that occur in class library calls.
- Determine why exceptions are occurring by examining the information that is passed back in the exception class object.

Throwing Your Own Exceptions Using the Exception Classes

In addition to catching exception class exceptions that are thrown by class library functions, you can also throw them in your own code. Throwing exception class exceptions in your own code has the following advantages:

- The exception classes provide a convenient package for exception information.
- If you use one of the predefined exception classes or derive one of your own from `IException`, you can use the same `catch` statement to catch exceptions that are generated by both class library functions and your own functions.

Consider the following simple example. The `getFirstChar` function calls the `IASSERTSTATE` macro with a `get` call as an argument. If the `get` call fails, it returns zero and the `IASSERTSTATE` macro throws an `IInvalidRequest` exception.

```
//   Using the IASSERTSTATE macro

   #include <iostream.h>
   #include <fstream.h>
   #include <iexcept.hpp>

   void openFile(fstream& fs, char *filename){
      fs.open(filename, ios::in);
   }

   char getFirstChar(fstream& fs) {
      char c;
      IASSERTSTATE(fs.get(c));
      return c;
   }

   void main() {
      char c;
      char * filename = "source.dat";
      fstream fs;
      openFile(fs, filename);
      try {
         c = getFirstChar(fs);
         cout << "Here is first character: " << c << endl;
      }
      catch(IException &ie)
      {
         cout << "Type of exception is: " << ie.name() << endl;
         cout << "Location of exception is: "
              << ie.locationAtIndex(0)->fileName() << endl;
         if (ie.isRecoverable())
            cout << "Exception is recoverable" << endl;
         else
            cout << "Exception is unrecoverable" << endl;
      }
   }
```

Suppose that this example is run, and the source.dat file is not available. The call to open in the OpenFile function will fail. When getFirstChar is called within the try block, an exception will be thrown by the IASSERTSTATE macro. This exception will be caught by the catch statement in main, and the output will look something like this:

```
Type of exception is: IInvalidRequest
Location of exception is: iopen.C
Exception is recoverable
```

Macros Used with the Exception Classes

The exception classes support a set of macros that allow you to manage the exception classes conveniently. You can use these macros to throw exceptions and to declare and define subclasses of IException or one of its subclasses.

ITHROW

Accepts as input an object of any IException subclass. It expands to add the location information to the instance, logs all instance data, and then throws the exception.

IRETHROW

Accepts as input a predefined instance of any subclass of IException that has been previously thrown and caught. Like the ITHROW macro, it also captures the location information, and logs all instance data before rethrowing the exception.

IASSERTSTATE

This macro accepts an expression to be tested as input. The expression is *asserted* to be true, meaning that you anticipate that it is true and are stating so to the compiler. If it evaluates to false, it invokes the IExcept__assertState function, which creates an IInvalidRequest exception. Location information is added to the exception, which is then logged and thrown.

IASSERTPARM

This macro accepts an expression to be tested as input. The expression is asserted to be true. If it evaluates to false, it invokes the IExcept__assertParameter function, which creates an IInvalidParameter exception. Location information is added to the exception, which is then logged and thrown.

IEXCLASSDECLARE

Creates a declaration for a subclass of IException or one of its subclasses.

IEXCLASSIMPLEMENT

Creates a definition for a subclass of IException or one of its subclasses.

IEXCEPTION_LOCATION
Expands to create an instance of the IExceptionLocation class.

INO_EXCEPTIONS_SUPPORT
Provided in support of compilers that lack exception handling implementation. If it is defined, the ITHROW macro ends the program after capturing the location information and logging it, instead of throwing an exception. This macro may not work correctly on all compilers.

ITHROWGUIERROR
This macro takes as its only argument the name of the GUI function that returned an error code. It calls the IGUIError::throwGUIError function, which creates an IGUIError instance and uses it to create an IAccessError instance, adds location information, logs out the exception data, and throws the exception. The exception severity is set to recoverable. Only used this macro if the error information that is retrievable by the IGUIErrorInfo class is available.

ITHROWGUIERROR2
This macro takes three arguments:

- The name of the GUI function that returned an error code
- One of the values of the IErrorInfo::ExceptionType enumeration, which indicates the type of exception to be created
- One of the values of the IException::Severity enumeration, which indicates the severity of the exception

Only use this macro if the error information that is retrievable by the IGUIErrorInfo class is available.

ITHROWSYSTEMERROR
This macro takes four arguments:

- The error ID returned from the system function
- The name of the system function that returned an error code
- One of the values of the IErrorInfo::ExceptionType enumeration, which indicates the type of exception to be created
- One of the values of the IException::Severity enumeration, which indicates the severity of the exception

Why Use the Macros?

You can manage exceptions that are objects of the exception classes directly. You can call member functions directly to create objects, and query and set their values. You can also explicitly derive your own classes from the existing exception classes. Often, however, it is more convenient to use the macros provided by the exception classes.

Exception Classes Macros

Consider the example that used the IASSERTSTATE macro:

```
//   Using the IASSERTSTATE macro

   #include <iostream.h>
   #include <fstream.h>
   #include <iexcept.hpp>

   void openFile(fstream& fs, char *filename){
      fs.open(filename, ios::in);
   }

   char getFirstChar(fstream& fs) {
      char c;
      IASSERTSTATE(fs.get(c));
      return c;
   }

   void main() {
      char c;
      char * filename = "source.dat";
      fstream fs;
      openFile(fs, filename);
      try {
         c = getFirstChar(fs);
         cout << "Here is first character: " << c << endl;
      }
      catch(IException &ie)
      {
         cout << "Type of exception is: " << ie.name() << endl;
         cout << "Location of exception is: "
              << ie.locationAtIndex(0)->fileName() << endl;
         if (ie.isRecoverable())
            cout << "Exception is recoverable" << endl;
         else
            cout << "Exception is unrecoverable" << endl;
      }
   }
```

This code could be rewritten to invoke the exception class member functions directly:

```
//   Invoking the IException member functions directly

   #include <iostream.h>
   #include <fstream.h>
   #include <iexcept.hpp>

   void openFile(fstream& fs, char *filename){
      fs.open(filename, ios::in);
   }

   char getFirstChar(fstream& fs) {
      char c;
      if (!fs.get(c)) {
         IInvalidRequest ir(" ", 0, IException::recoverable);
         IExceptionLocation il("imac.C","getFirstChar",5);
         ir.addLocation(il);
         throw(ir);
      }
      return c;
   }

   void main() {
      char c;
      char * filename = "source.dat";
      fstream fs;
      try {
         c = getFirstChar(fs);
         cout << "Here is first character: " << c << endl;
      }
      catch(IException &ie)
      {
         cout << "Type of exception is: " << ie.name() << endl;
         cout << "Location of exception is: "
              << ie.locationAtIndex(0)->fileName() << endl;
         if (ie.isRecoverable())
            cout << "Exception is recoverable" << endl;
         else
            cout << "Exception is unrecoverable" << endl;
      }
   }
```

Notice how the single IASSERTSTATE in the getFirstChar function is replaced with a test of the return value of get, the definition of an IInvalidRequest object, the definition of an IExceptionLocation object, and an explicit throw statement. You can see that the version of the program that uses the IASSERTSTATE macro is simpler and easier to code.

Using the ITrace Class

The ITrace class provides a set of facilities that allow you to put trace statements in your code conveniently. The most convenient way to use ITrace is through the macros that it supports.

Using the Trace Macros to Control Trace Output

The ITrace class is convenient to use because it allows you to turn trace statements on and off easily. By defining certain macros and by using the macros in the ITrace class to create trace output, you can selectively turn tracing on and off. There are three special trace macros:

- IC_TRACE_RUNTIME
- IC_TRACE_DEVELOP
- IC_TRACE_ALL

By defining or not defining these macros, you can specify whether or not the trace macros are expanded, and thus whether or not your program produces trace output.

If IC_TRACE_RUNTIME is defined, the following macros are expanded:

IMODTRACE_RUNTIME

This macro takes one argument that is the name of the current module. It creates an ITrace object using the module name as the name of the trace and the current line number as the line number.

`IFUNCTRACE_RUNTIME`
This macro takes no arguments. It creates an `ITrace` object using the function name as the name of the trace and the current line number as the line number.

`ITRACE_RUNTIME`
This macro takes a single argument. This argument is written to the trace location.

If `IC_TRACE_DEVELOP` is defined, all of the macros that are expanded when `IC_TRACE_RUNTIME` is defined, are also expanded. In addition, the following macros are expanded:

`IMODTRACE_DEVELOP`
This macro takes one argument. Typically you use the argument to name the current module. This macro creates an `ITrace` object using the module name as the name of the trace and the current line number as the line number.

`IFUNCTRACE_DEVELOP`
This macro takes no arguments. It creates an `ITrace` object using the function name as the name of the trace and the current line number as the line number.

`ITRACE_DEVELOP`
This macro takes a single argument. This argument is written to the trace location.

If `IC_TRACE_ALL` is defined, all of the trace macros are expanded.

An Example of Using ITrace

The following piece of code shows one way that you could use the trace macros to produce trace output for your programs. In this code, the macros `IFUNCTRACE_DEVELOP` and `ITRACE_DEVELOP` are used to create trace statements that indicate that the flow of control has passed through the functions `openFile` and `getFirstChar`.

```
// Producing trace output with the ITrace class

#define IC_TRACE_DEVELOP

#include <iostream.h>
#include <fstream.h>
#include <iexcept.hpp>
#include <itrace.hpp>

void openFile(fstream& fs, char *filename){
   IFUNCTRACE_DEVELOP();
   fs.open(filename, ios::in);
   ITRACE_DEVELOP("after open statement");
}
```

```
char getFirstChar(fstream& fs) {
   char c;
   IFUNCTRACE_DEVELOP();
   fs.get(c);
   ITRACE_DEVELOP("after get statement");
   return c;
}

void main() {
   char c;
   char * filename = "source.dat";
   fstream fs;
   //
   // static functions to enable tracing and direct
   // tracing output to standard output
   //
   ITrace::enableTrace();
   ITrace::writeToStandardOutput();
   openFile(fs, filename);
   c = getFirstChar(fs);
   cout << "Here is first character: " << c << endl;
}
```

Notice that, in this code, the static functions `enableTrace` and `writeToStandardOutput` are used to enable tracing and to direct the trace output to standard output.

Because the macro `IC_TRACE_DEVELOP` is defined, the trace macros produce trace output. In addition, the trace output has been explicitly directed to standard output, so the output of the code looks like this:

```
+openFile(fstream&,char*)
  >after open statement
-openFile(fstream&,char*)
+getFirstChar(fstream&)
  >after get statement
-getFirstChar(fstream&)
Here is first character: t
```

Suppose that you wanted to turn off the trace output in this program. One way to do it is to modify the code so that the macro `IC_TRACE_DEVELOP` is not defined. If you do this, the trace macros are not expanded, and no trace output is produced. The output of this code with `IC_TRACE_DEVELOP` not defined looks like this:

```
Here is first character: t
```

Using the ITrace Class

19

Date and Time Classes

The IDate and ITime classes provide you with data types to store and manipulate date and time information. With these classes, you can create date and time objects, and use member functions to do the following:

- Write date and time objects to an output stream
- Access detailed information about dates or times
- Compare dates or times
- Test the characteristics of date or time objects
- Add or subtract days from a date, or hours, minutes, or seconds from a time
- Convert between date formats or between time formats.

The IDate and ITime classes are independent. When an ITime object's time passes 23:59:59 (24-hour format) or 11:59:59 p.m. (12-hour format), it has no effect on the value of any IDate object. If you want to have interdependent date and time objects you must create your own class, containing IDate and ITime data members, and define constructors, operators, and member functions that take into account the dependency of the IDate and ITime data members. See “Simple Combined Date and Time Example” on page 229 for an example of how to do this.

IDate Class

The IDate class uses Gregorian calendar dates. The Gregorian calendar is in general use and consists of the 12, months January to December.

IDate also supports the Julian date format, which contains the year in positions 1 and 2, and the day of the year in postions 3 through 5. If the day of the year is less than three digits, zeros are added on the left to increase the size to three digits. For example, February 14, 1965 is 65045 as a Julian date. (February 14 is the 45th day of the year.)

The IDate class returns the names of the days and months in the language defined by the current locale. For information on defining the locale, see the standard C library function setlocale().

Creating an IDate Object

You can create an IDate object using different IDate constructors. For example:

```
IDate OneDay(IDate::June,30,1994);      // Month, day, year
IDate AnotherDay(23,IDate::April,1961); // Day, month, year
IDate SomeDay(940616);                  // Julian date format
IDate Yesterday(1994,177);              // Year, day of year
```

The constructors accepting a month use the IDate enumeration Month, whose members are named January through December (the months of the year in English).

Changing an IDate Object

You can add days to, or subtract days from, an IDate object. You can also subtract one date from another, in which case the result is the number of days between the two dates. For example:

```
IDate Day1, Day2;
int NumDays;
Day1=IDate::today();
Day2=Day1+1;        // Day2 is one day after Day1
Day2+=2;            // Day2 is now three days after Day1
NumDays=Day2-Day1;  // NumDays=3
```

Note that you cannot add two IDate objects together, because such an addition does not make sense. However, you can add two ITime objects together.

Information Functions for IDate Objects

The IDate class defines information functions that you can use to obtain specifics about an IDate object. For example, you can find out what day of the week, month, or year an IDate object's date falls on, or what the name of the day or month is for the current locale. You can also find out what today's date is. The following example shows some of the IDate information functions:

```
//  Information functions for IDate class

#include <iostream.h>
#include <istring.h>
#include <idate.h>

void main () {
   IDate Day1(27,IDate::May,1964);
   cout << Day1.dayName() << " "
        << Day1.monthName() << " "
        << Day1.dayOfMonth() << " out of "
        << IDate::daysInMonth(Day1.monthOfYear(), Day1.year()) << " days in month, "
        << IDate::daysInYear(Day1.year()) << " days in year "
        << Day1.year() <<'.' << endl;
        }
```

This program produces the following output:

```
Wednesday May 27 out of 31 days in month, 366 days in year 1964.
```

Testing and Comparing IDate Objects

You can compare two IDate objects to determine whether they are equal, or whether one is later than the other. The following operators are defined: ==, !=, <, <=, >, >=. For example, the expression if ((Day1>Day2) && (Day1!=Day3) evaluates to true if Day1 is January 1 1994, Day2 is June 3 1968, and Day3 is July 12 1941.

You can also check whether a particular year is a leap year, or whether a particular combination of day, month, and year is valid. The isLeapYear() function returns

True if its integer argument is a leap year. The isValid() function accepts combinations of day, month, and year (or day of year and year), and returns True if the provided date is valid. For example, it returns True for the first date below, and False for the second date:

```
if (IDate::isValid(IDate::June, 30, 1990)) // ...
if (IDate::isValid(1965,366)             // ... False (No day number 366 in 1965)
```

ITime Class

The ITime class refers to time in the 24-hour format by specifying time units (hours, minutes, seconds) past midnight. If you want to display ITime objects in the 12-hour format, you must convert them to IStrings using the asString function with a char* argument of "%r". (This argument is a format string. All format specifiers of the strftime() function of the standard C library are supported by the IString conversion function.)

Note: Objects of the ITime class are precise only up to the nearest second, and cannot be used for more precise timings.

Creating an ITime Object

You can create an ITime object and initialize it to a number of seconds past or before midnight, or to a number of hours, minutes, and optionally seconds past midnight:

```
ITime Time1(33556),     // 09:19:16
      // 33556 = 9 hours   (32400 seconds), 19 minutes (1140 seconds),
      // 16 seconds (adds up to 33556)
      Time2(-33556),    // 14:40:44
      // (9 hours, 19 minutes and 16 seconds BEFORE midnight)
      Time3(12,00),     // 12:00:00 (noon)
      Time4(3,3,3);     // 03:03:03
```

The constructors translate incorrect times into valid ITime objects using modulo arithmetic. For the seconds past midnight format, any number whose absolute value is greater than or equal to 86400 is divided by 86400, and the remainder is used to calculate the time. For the hours, minutes, and optional seconds format, excess minutes and seconds are added to the hours and minutes values, respectively, and if the hour exceeds 23 it is divided by 24 and the remainder is taken. For example:

```
ITime Time1(133556),     // 13:05:56 (13356-86400=47156 seconds after midnight)
      Time2(-133556),    // 10:54:04 (13356-86400=47156 seconds BEFORE midnight)
      Time3(10,119,60),  // 12:00:00 (noon) (10 hours plus 119 minutes plus 60 seconds)
      Time4(33,33);      // 09:33:00  (33 hours - 24 hours = 9 hours)
```

Changing an ITime Object

You can add or subtract two times. Four operators are provided: +, +=, -, and -=. The following example shows the use of these operators:

```
ITime Start(12:00), Duration(2:00),
      Done=Start+Duration;  // Done=14:00
Start=Done-Duration;        // Start=12:00 still
Start+=Duration;            // Start=14:00
Start-=Duration;            // Start=12:00 again
```

Information Functions for ITime Objects

Three of the information functions return an `ITime`'s hour, minute, or second settings; the other information function returns the current time as determined by the system clock. For example:

```
ITime Time1(ITime::now());
cout << Time1.hours() << " o'clock occurred "
     << Time1.minutes() << " minutes and "
     << Time1.seconds() << " seconds ago." << endl;
```

This displays a result such as the following:

```
12 o'clock occurred 16 minutes and 23 seconds ago.
```

Comparing ITime Objects

Functions are defined to let you compare `ITime` objects for equality, inequality, or relative position in time. The following operators are defined: `==`, `!=`, `<`, `<=`, `>`, `>=`. In the following example, a message is displayed if enough time elapses between the first and second calls to the `now()` member function:

```
#include <itime.hpp>
#include <iostream.h>
ITime First(ITime::now());
void main() {
   ITime Second=ITime::now();
   if (First<Second) // Some time has passed
      cout << "You must be debugging me!" << endl;
   }
```

This message usually does not print when the program is run outside of a debugging session. However, if you debug the program and step through each line slowly, the message may be displayed, because the first `ITime` object is initialized during program initialization (before **main** is called) while the second `ITime` object is initialized within **main**.

Writing an ITime Object to an Output Stream

`ITime` defines an output operator that writes an `ITime` object to an output stream in the format `hh:mm:ss`. If you want to write the object out in a different format, you should convert the object to an `IString` using the `asString` member function. This member function accepts a `char*` argument containing a format specifier. The format specifier is the same one as used by the C library function **strftime**. The following program displays some valid specifiers and the output they produce:

```
// Examples of ITime output

#include <istring.hpp>
#include <itime.hpp>
#include <iostream.h>
#include <iomanip.h>  // needed for setw(), to set output stream width

void main() {
   char* FormatStrings[]={
      "%H : %M and %S seconds", // %H, %M, %S - 2 digits for hrs/mins/secs
      "%r",                     // %r - standard 12-hour clock with am/pm
   "%T",                     // %T - standard 24 hour clock
   "%T %Z",                  // %Z - local time zone code
   "%1M past %1I %p"         // %1... - One digit for hour/minute
   };                        // %p - am/pm

cout.setf(ios::left,ios::adjustfield);    // Left-justify output

cout << setw(30) << "Format String"       // Title text
     << setw(40) << "Formatted ITime object" << endl;

for (int i=0;i<5;i++) {                   // Show each time
   IString Formatted=ITime::now().asString(FormatStrings[i]);
   cout << setw(30) << FormatStrings[i]
        << setw(40) << Formatted << endl;
   }
}
```

The program produces output that looks like the following:

```
Format String                 Formatted ITime object
%H : %M and %S seconds        16 : 13 and 04 seconds
%r                            04:13:04 PM
%T                            16:13:04
%T %Z                         16:13:04 EST
%1M past %1I %p               13 past 4 PM
```

Simple Combined Date and Time Example

The following example shows a class MyDateTime that links its date and time data members together within its addition operator definition. The class has three data members, one from each of IDate and ITime, and one for the number of days to be added when the addition operator is used. The class defines two addition operators: one that accepts another MyDateTime object, and uses the number of days and the time as the basis for the addition; and one that accepts only an ITime object, and adds that to the MyDateTime object's time. Both addition operators check for wraparound in the ITime member, and increment the IDate member if wraparound has occurred.

```
// Simple combined date-time class
#include <idate.hpp>
#include <itime.hpp>
#include <iostream.h>

class MyDateTime {
  public:
   IDate date;          // date subobject
   ITime time;          // time subobject
   int addDays;         // number of days to add for addition operator

   // Copy constructor
   MyDateTime(IDate adate, ITime atime, int add = 0):
      addDays(add), date(adate), time(atime) {}

   // Default constructor
   MyDateTime() {}

   // Addition operator for other MyDateTime objects
   MyDateTime operator + ( const MyDateTime &aDateTime) const {
      MyDateTime temp;

      // Add any addDays value to date if necessary
      temp.date = this->date + aDateTime.addDays;
      // Add times together
      temp.time = this->time + aDateTime.time;

      // If resulting time is greater than original time,
      // clock wrapped around, so increment date
      if (temp.time < this->time) temp.date += 1;

      return temp;
      }

   // Addition operator for ITime objects
   MyDateTime operator + ( const ITime &time) const {
      MyDateTime temp;
      temp.date = this->date;

      // Add time to time member of MyDateTime temporary
      temp.time = this->time + time;

      // If resulting time is greater than original time,
      // clock wrapped around, so increment date
      if (temp.time < this->time) temp.date += 1;

      return temp;
      }
   };

ostream& operator << (ostream& os, MyDateTime& dt) {
   cout << dt.date << " at " << dt.time;
   return os;
   }

void main() {
   MyDateTime TodayNow(IDate::today(), ITime::now()),
              Add, Temp;
   Add.addDays = 17;
   Add.time = ITime(12,24);
   Temp = TodayNow + Add;
   cout << "Right now it is " << TodayNow << ".\n"
        << "In 17 days, 12 hours, and 24 minutes, it will be "
        << Temp << "." << endl;

   Temp = TodayNow + ITime(21,16);
   cout << "In 21 hours and 16 minutes, it will be " << Temp << "." << endl;
   }
```

This program produces output that resembles the following:

```
Right now it is 06/22/94 at 13:25:31.
In 17 days, 12 hours, and 24 minutes, it will be 07/10/94 at 01:49:31.
In 21 hours and 16 minutes, it will be 06/23/94 at 10:41:31.
```

Part 5. The Database Access Class Library

Database Access

20 Using the Database Access Class Library

This section describes how to use the generated classes and the library classes to access data in DB2/2 tables.

When you used the Data Access Builder tool, you had a choice to generate either **Parts** or **IDL**, or both.

Now you use the generated code and classes to access data in the DB2/2 database. This can be done using any of the following:

- Visual Builder program with the generated part files,
- C++ program with the generated part files, or
- SOM program with the generated IDL files.

The steps to access the database table are the same regardless of which type of program you are writing.

The steps are:

1. Compile the generated files
2. Access the data in the DB2/2 table
   a. Connect to the database
   b. Accessing database tables
   c. Commit or rollback your transactions
   d. Disconnect from the database.

Using Visual Builder Programs

Compiling a Database Part

You can build a database part using either of the following:

- From a project

  If you start the DAX tool from a project, the makefile is not generated. Use the project **Build** action to build the part.
- From an OS/2 window

  If you start the DAX tool from a project, the makefile is generated for you. Issue an **nmake** from the command line to start the make.

The generated makefile or a project build contain steps to:

1. Invoke **icc** to compile the *filenamey.cpp* source file
2. Invoke **sqlprep** to pre-compile the *filenamev.sqc* file. The application is pre-compiled with binding enabled (/P) to generate a package file which is stored in the same database where the table (that the class maps to) came from. For more information on sqlprep and package considerations, see the *Database 2 for OS/2 Application Programming Guide*. This produces a *filenamev.c* file and a *filenamev.bnd* file.
3. Invoke **icc** to compile the *filenamev.c* file into a *filenamev.obj* module.
4. Invoke **icc** to link the .obj files produced in step 1 and 3 to create a *filenamev.dll* file. The *filenamev.def* file is used in building the .dll.
5. Invoke implib to create a *filenamev.lib* file.

Building a Part from a Project

1. Open the **VisualAge C++ 3.0 Tools** folder.
2. Copy the **DAXSAMP Database DLL** project from the **Database folder** in the Samples folder of **VisualAge C++**. The project should be copied with a new project name.
3. Open the new project.
4. Choose **Settings** from the **View** menu and fill in the appropriate target and location (directories) information for the new project.
5. The SQLPrep, compile, link and import library actions are set appropriately to create the DLL. You need to set the LIB and INCLUDE environment variable to reference this new project target for any projects that use the DLL. You also need to copy the DLL produced by this project to a directory on your LIBPATH. If you generate more than one class for the DLL, you need to create a .def file. Use the ones generated as an example.
6. Start **Data access** from the project pull-down menu and generate your mapping.
7. Exit Visual Builder
8. Choose **Build** from the **Project** pull-down menu.

Building a Part from an OS/2 Window

To compile a database part once it has been generated:

1. Exit Data Access Builder and open an OS/2 command window
2. Ensure that your paths are set up properly in your *config.sys* file.
3. Make your current directory the directory where your generated files reside.
4. From the OS/2 command line, type:

   ```
   nmake -f filenamev.mak
   ```

   where `filename` is the name that you assigned to the class.

Using Database Parts in Visual Builder

Once you have compiled the part, you are ready to load the part file into the Visual Builder. The part file name for the Data Access Builder is *VBDAX.VBB* and is in the *DDE4VB* directory. For a generated part, import the Visual Builder part information file (*filename*V.VBE) separately. For more information, see Chapter 21, “Constructing Applications Using Data Access Builder and the Visual Builder” on page 255.

To access a database table in a Visual Builder application, add the IDatastore part from *VBDAX*. You link command buttons with the connect, transact, disconnect and all the data access operations of the generated classes. For those operations that require parameters, use entry fields.

Accessing the DB/2 Table

There are two ways you might want to connect to a database. The first method would be for applications where it is not necessary for the user to know or modify the database name, while the second is used when more flexibility is required for accessing databases, changing user logons, etc.

In simple case, you would use the non-visual IDatastore part with settings appropriate for your application.

The second method provides more flexibility - use the IDSConnectCanvas (or a similar part of your own design), and allow the user to fill in the database name, userid and password. The IDatastore interface is also exposed on IDSConnectCanvas, so this part can be used in your application just as you use IDatastore. The settings of IDSConnectCanvas can be used to set the defaults for the connection.

IDatastore and IDSConnectCanvas expose attributes, actions and events you can use to control the connection. These interfaces are enabled with the notification framework, so they can be used with applications developed using the Visual Builder. Visual Builder attribute to attribute connections can be used to change the attributes of the connection.

Attributes

datastoreName Name of the datastore for this connection. You may initialize this in the settings of the part.

userName Name of the user attaching to the datastore. You may initialize this in the settings of the part.

authentication Password of the user attaching to the datastore. You may initialize this in the settings of the part.

isConnected Returns a Boolean value depending on the connect state. You can also use an attribute to attribute connections which allow, for example, enabling a Disconnect button once a connection is established.

shareModeExclusive Only one process at a time may access a database with shareModeExclusive enabled. You may initialize this in the settings of the part.

Actions

connect Connect to the database using the current IDatastore settings. A disconnect is performed if a current connection exists (with a commit or rollback depending on your system defaults). You can only be connected to one database at a time. If the userName specified in that attribute is not currently logged on, a logon is performed. If the logon fails, an exception is thrown.

disconnect Disconnect any current connection to the database. If the user was logged on during the connect action, that user is then logged off.

commit Commits the current transaction.

rollback Rollback (cancel) the current transaction.

Events

Connected Signaled when a connection is established. You could use this event to show a second application window.

Disconnected Signaled when a connection is terminated. You could use this event to hide a second application window.

Transacted Signaled when a transaction (commit or rollback) is completed. You could use this event to refresh the information in an application window.

Each of the attributes will also signal events when they are modified.

Accessing Data in the DB/2 Table

You use the parts generated by Data Access to access the data in your database.

First, use the database connection part to connect to the database (see “Accessing the DB/2 Table” on page 235).

Then, use the parts that are generated by the Data Access Builder to access the information in the tables.

For example, classes *Car* and *CarManager* would be generated from a table called *Car*.

You would use the *Car* class to manipulate individual rows in the Car table. After importing the generated file, carv.vbe,
into the Visual Builder, you would find the following actions, attributes and methods:

Attributes

Columns Each mapped column from the table would be an attribute of the generated class. A method is provided to get and set the value of each attribute, as well as checking or setting the attributes as Null (if allowed for that column). Attributes are enabled for notification, and will also return IString() representations. Thus attributes can be directly connected to other Visual Builder parts. Each part will then reflect changes to the other.

isReadOnly Boolean representing whether the object is read only.

isDefaultReadOnly Boolean representing whether the table is read only as defined in the database.

Actions

retrieve You can retrieve a row from the table by first setting the data identifier (key) attributes of the object, then calling the retrieve method. All of the attributes of this object will then be set to the current values found for the row with that key in the table. An exception will be thrown if the row does not exist, or if more than one row with that data identifier is found.

add Set the object attributes with the desired values. Then call this method. A row will be added to the table with the columns reflecting the attribute values. An exception will be thrown if the database detects that a row already exists with a key value matching that set in the attributes representing the data identifier. An exception will be thrown if the table is read only.

update Retrieve a row into the object using the retrieve() method. You can then update the values, and update that row in the database with this method. Any object with a key (or a column value) matching the attribute values of the data identifier will be updated, so you should ensure that the data identifier is unique. An exception will be thrown if the table is read only.

del Any object with a key (or a column value) matching the attribute values of the data identifier will be deleted. An exception will be thrown if the table is read only.

asString Will return an IString that contains the concatenated IString representations of the attributes making up the data identifier.

setReadOnly Can be used to set an object as read only. Exceptions will be thrown on any attempt to update the object.

Events Each of the attributes of the class will signal events when they are modified.

You would use the *CarManager* class to retrieve multiple rows in the Car table. After importing the generated file, carv.vbe, into the Visual Builder, you would find the following actions, attributes and methods:

- Attributes:

  Items Items is a sequence of pointers to the parts retrieved in a refresh or select action. You can connect this directly to the Items attributes of other Visual Builder parts, such as list boxes or containers.

- Actions:

  refresh Retrieves all items in the table and stores them in the Items sequence.

  select Retrieves all items in the table matching the specified clause and stores them in the Items sequence.

- Events: Items signals an event whenever it is modified (for example, through a refresh or select action).

Using C++ Programs

Depending on whether you created source code for a part or for IDL, now you need to compile the source code.

Compiling a Database Part

You can build a database part using either of the following:

- From a project

  If you start the DAX tool from a project, the makefile is not generated. Use the project **Build** action to build the part.
- From an OS/2 window

  If you start the DAX tool from a project, the makefile is generated for you. Issue an **nmake** from the command line to start the make.

The generated makefile or a project build contain steps to:

1. Invoke **icc** to compile the *filenamey.cpp* source file
2. Invoke **sqlprep** to pre-compile the *filenamev.sqc* file. The application is pre-compiled with binding enabled (/P) to generate a package file which is stored in the same database where the table (that the class maps to) came from. For more information on sqlprep and package considerations, see the *Database 2 for OS/2 Application Programming Guide*. This produces a *filenamev.c* file and a *filenamev.bnd* file.

3. Invoke **icc** to compile the *filenamev.c* file into a *filenamev.obj* module.
4. Invoke **icc** to link the .obj files produced in step 1 and 3 to create a *filenamev.dll* file. The *filenamev.def* file is used in building the .dll.
5. Invoke implib to create a *filenamev.lib* file.

Building a Part from a Project

1. Open the **VisualAge C++ 3.0 Tools** folder.
2. Copy the **DAXSAMP Database DLL** project from the **Database folder** in the Samples folder of **VisualAge C++**. The project should be copied with a new project name.
3. Open the new project.
4. Choose **Settings** from the **View** menu and fill in the appropriate target and location (directories) information for the new project.
5. The SQLPrep, compile, link and import library actions are set appropriately to create the DLL. You need to set the LIB and INCLUDE environment variable to reference this new project target for any projects that use the DLL. You also need to copy the DLL produced by this project to a directory on your LIBPATH. If you generate more than one class for the DLL, you need to create a .def file. Use the ones generated as an example.
6. Start **Data access** from the project pull-down menu and generate your mapping.
7. Exit Visual Builder
8. Choose **Build** from the **Project** pull-down menu.

Building a Part from an OS/2 Window

To compile a database part once it has been generated:

1. Exit Data Access Builder and open an OS/2 command window
2. Ensure that your paths are set up properly in your *config.sys* file.
3. Make your current directory the directory where your generated files reside.
4. From the OS/2 command line, type:

   ```
   nmake -f filenamev.mak
   ```

   where `filename` is the name that you assigned to the class.

The makefile contains steps to:

1. Invoke **icc** to compile the *filenamey.cpp* source file
2. Invoke **sqlprep** to pre-compile the *filenamev.sqc* file. The application is pre-compiled with binding enabled (/P) to generate a package file which is stored in the same database where the table (that the class maps to) came from. For more information on sqlprep and package considerations, see the *Database 2 for OS/2 Application Programming Guide*. This produces a *filenamev.c* file.
3. Invoke **icc** to compile the *filenamev.c* file into a *filenamev.obj* module.
4. Invoke **icc** to link the .obj files produced in step 1 and 3 to create a *filenamev.dll* file. The *filenamev.def* file is used in building the .dll.
5. Invoke implib to create a *filenamev.lib* file.

Accessing the Data in the DB2/2 Table

Once you have compiled the part, you are ready to connect to the database. You can only be connected to one database at a time.

To access the database, do the following:

1. Connect to the database
2. Access the database tables
3. Commit or rollback
4. Disconnect from the database

Connect to the Database

To connect to the database:

1. Include the idsmcon.hpp header file.
2. Create object from IDatastore.
3. Establish a connection to the database using the IDatastore object.

Here is an example:

```
... ...
#include "idsmcon.hpp"

// create an object of IDatastore
IDatastore   dsm("DSNAME","USERID","PASSWORD");

// establish the connection to the datastore
dsm.connect();
... ...
```

Accessing Database Tables

Once the connection to the database has been established, the data in the tables can be accessed.

You access data in a table using the table-class mappings you created with the Data Access Builder tool. The mapping indicates the table columns that are used when the database is accessed.

After you created the table-class mappings, you generated the source files. When you generated the source files, **two** classes were generated for accessing data in the table.

The first class accesses a single row of data. This class is derived from the IPersistentObject class. With this class, you can add, delete, retrieve or update a single row using a data identifier. In this case, the data identifier must be set *before* invoking the method.

The other class accesses multiple rows of data. This class is identified by a suffix of “Manager” to the class name. For example, if Car is the first class, CarManager is the other class. This class is derived from the IPOManager class. With this class, you can retrieve multiple rows using the refresh or select methods. The refresh method provides a snapshot of a complete table. The select method provides a snapshot of a selected set of a table.

Note: When updating rows in a table, to avoid unexpected values written to other columns of the row, it is recommended to retrieve the row before updating it.

Mappings to different columns in the same table require different methods. For example, if you have a table containing inventory information, you can create one mapping for all inventory information, called *Inventory*, and a second mapping for only the price list information, called *PriceList*. In this case, four classes are generated, two for each mapping:

1. Inventory
2. InventoryManager
3. PriceList
4. PriceListManager

A header file is generated for each mapping. You must include them in your program. Since only the first seven characters are used for the name, in this example, the header names are:

- inventov.hpp
- priceliv.hpp

You can change the filename on the Names page of the class notebook.

With these include files you create objects from the classes and access the data in the database.

For example:

```
// ... ...
// include the generated header
#include "inventov.hpp"
#include "priceliv.hpp"

// ... ...
// create objects from the class
Inventory          invObject;
InvenotryManager   invMgrObject;
PriceList          prcLstObject;
PriceListManager   prcLstMgrObject;

// ... ...
// to add a row where PRODNUMBER is the primary key of the table
invObject.setProdNumber(1234);
invObject.setDescription("IBM PS1");
invObject.setOnHandQuantity(30);
invObject.setAverageCost(1000.00);
invObject.setListingPrice(2999.99);
invObject.add();

// To update a row, specify the key and the new value. This
// example changes the listing price only.
prcLstObject.setProdNumber(1234);
prcLstObject.retrieve();
prcLstObject.setProdNumber(1234);
prcLstObject.setListingPrice(1999.99);
prcLstObject.udpate();

// to delete a row
invObject.setProdNumber(1234);
invObject.del();
```

Commit Work or Roll Back

Use the IDatastore object to do commit or rollback the work you have done.

Here is an example to commit the transaction:

```
// to commit ...
dsm.commit();
```

Here is an example to roll back the transaction:

```
// to rollback ...
dsm.rollback();
```

Disconnecting from the Database

You must disconnect from one database before you can connect to another. Disconnect using the disconnect method of the IDatastore object. Here is an example:

```
// disconnect the connection
dsm.disconnect();
```

Read Only Support Considerations

Read only support has the following member functions:

1. isReadOnly,
2. isDefaultReadOnly,
3. setReadOnly.

Read only status can be checked by calling isDefaultReadOnly. If the part is generated from a read only view, the function returns true. In this case, the update, delete, and add functions always return an exception.

If the part is not generated from a read only view, the function returns false. You can update, delete, or add the table (or view) using the class. However, you can make it a read only one by calling the setReadOnly function with true value. You check if the part set to read only using the isReadOnly function.

If the isDefaultReadOnly function results true, you will have an exception when calling the setReadOnly function with false.

Here is the sample code for the read only support:

```
// create car object
Car           CarObject;

// Check the defaultReadOnly status before delete a row.
if (CarObject.isdefaultReadOnly() == false) {
   // Delete a row with the key value, license = 'ABC-456'
   carObject.setLicense("ABC-456");
   carObject.del();
}

// set the currentReadOnly to true
CarObject.setReadOnly(true);
```

Null Value Support Consideration

There are 3 member functions for each attribute to support null value. If the attribute name is model. The member functions are isModelNullable, isModelNull, and setModelToNull.

To check if model is a nullable attribute, you use isModelNullable. If the result is true (nullable), you can use setModelToNull to make Model null. If the result is false, it is an exception to do so. You can also use isModelNull to check if model contains null.

Please note that once you use set an attribute to null, the value kept in the model is ignored for the data access operations. For example, the add operation will put NULL to the model column when it adds a row to the table.

Here is the sample code for the null value support:

```
// create car object
Car           CarObject;
boolean       result;

// Check if the model column is a nullable one
if (CarObject.isModelNullable() == true) {
   //model is nullable, we can set it to null
   carObject.setModelToNull(true);
}

// Check if model is set to null
result = CarObject.isModelNull();
```

Exception Handling

All exception classes in the database access classes are derived from IException. Here is an example of catching an exception in connecting to a datastore:

```
try {
  dsm.connect();
} catch (IDatastoreLogonFailed &exc) {
  cout << "ErrorId: " << exc.errorId() << endl;
  for (unsigned long i = 0; i < exc.textCount(); i++)
    cout << "Error Text: " << exc.text(i) << endl;
} catch(...) {
  cout << "Unknown exception occurs for dsm.connect() " << endl;
}
```

The classes in the generated files throws exception IDSAccessError only. The errorId indicates the type of the exception. If the error is an SQL error, use the getSqlca() function to access the SQLCA information. If the error is not an SQL error, the value returned by the getSqlca() function is not defined. Here is an example:

```
try {

} catch (IDSAccessError &exc) {
  cout << "ErrorId: " << exc.errorId() << endl;
  for (unsigned long i = 0; i < exc.textCount(); i++)
    cout << "Error Text: " << exc.text(i) << endl;
  if (exc.errorId() == DAX_ADD_SQLERR) {
    cout << "SQLCODE: " << exc.getSqlca().sqlcode << endl;
    cout << "SQLSTATE: " << exc.getSqlca().sqlstate << endl;
  }
} catch(...) {
  cout << "Unknown exception occurs for table1.add() " << endl;
}
```

Using SOM Programs

There are a number of IDL files kept in the IBMCLASS directory. They are the IDLs for the SOM classes. To use the SOM classes in C++, you have to generate the C++ usage binding files (.xh files). The IDLs generated by the Data Access Builder tool depend on these .xh files.

To generate the files, issue the following command in the IBMCLASS directory:

```
sc -sxh *.idl
```

To use the SOM classes in C programs, you have to generate C usage binding files (.h files) and issue the following command:

```
sc -sh *.idl
```

In additon, you need to generate the C usage binding files for the generated IDL. For example, if the generated IDL is cari.idl, you have to issue:

```
sc -sh cari.idl
```

A cari.h file is generated. This header file will be included in you C program.

Compiling the IDL

You can build the IDL using either of the following:

- From a project

  If you start the DAX tool from a project, the makefile is not generated. Use the project **Build** action to build the part.
- From an OS/2 window

  If you start the DAX tool from a project, the makefile is generated for you. Issue an **nmake** from the command line to start the make.

The generated makefile or a project build contain steps to:

1. Invoke **icc** to compile the *filenamey.cpp* source file
2. Invoke **sqlprep** to pre-compile the *filenamev.sqc* file. The application is pre-compiled with binding enabled (/P) to generate a package file which is stored in the same database where the table (that the class maps to) came from. For more information on sqlprep and package considerations, see the *Database 2 for OS/2 Application Programming Guide*. This produces a *filenamev.c* file and a *filenamev.bnd* file.
3. Invoke **icc** to compile the *filenamev.c* file into a *filenamev.obj* module.
4. Invoke **icc** to link the .obj files produced in step 1 and 3 to create a *filenamev.dll* file. The *filenamev.def* file is used in building the .dll.
5. Invoke implib to create a *filenamev.lib* file.

Building the IDL from a Project

1. Open the **VisualAge C++ 3.0 Tools** folder.
2. Copy the **DAXSAMP Database DLL** project from the **Database folder** in the Samples folder of **VisualAge C++**. The project should be copied with a new project name.
3. Open the new project.
4. Choose **Settings** from the **View** menu and fill in the appropriate target and location (directories) information for the new project.
5. The SQLPrep, compile, link and import library actions are set appropriately to create the DLL. You need to set the LIB and INCLUDE environment variable to reference this new project target for any projects that use the DLL. You also need to copy the DLL produced by this project to a directory on your LIBPATH. If you generate more than one class for the DLL, you need to create a .def file. Use the ones generated as an example.
6. Start **Data access** from the project pull-down menu and generate your mapping.
7. Exit Visual Builder
8. Choose **Build** from the **Project** pull-down menu.

Using SOM Programs

Building a IDL from an OS/2 Window

To compile the IDL once it has been generated:

1. Exit Data Access Builder and open an OS/2 command window
2. Ensure that your paths are set up properly in your *config.sys* file.
3. Make your current directory the directory where your generated files reside.
4. From the OS/2 command line, type:

   ```
   nmake -f filenamei.mak
   ```

   where `filename` is the name that you assigned to the class.

The makefile contains steps to:

1. Invoke the SOM compiler (**sc**) to generate the usage binding file, *filename.xh*, and the implementation binding file, *filename.xih*, for this som class.
2. Invoke **icc** to compile the *filenamex.cpp* source file. Compiling the *filenamei.cpp* file requires the *filename.xh* and *filename.xih* binding files generated by the SOM compiler in step 1.
3. Invoke **sqlprep** to pre-compile the *filenamei.sqc* file. The application is pre-compiled with binding enabled (/P) to generate a package file which is stored in the same database where the table (that the class maps to) came from. For more information on sqlprep and package considerations, see the *Database 2 for OS/2 Application Programming Guide*. This produces a *filenamei.c* file.

The *filename.obj* and *Sfilename.obj* are created. You can link them in with your application.

Accessing the Data in the DB2/2 Table

Once you have compiled the source code, you are ready to connect to the database. You can only be connected to one database at a time.

To access the database, do the following:

1. Connect to the database
2. Access the database tables
3. Commit or rollback
4. Disconnect from the database

Connect to the Database

To establish a connection, you must have the sdsmcon.xh header file.

To connect to a database:

1. Create Datastore class object from DatastoreFactory meta-class.
2. Create object from the Datastore class object.
3. Invoke the connect method to establish a connection

Here is the C++ code sample:

```
Environment *ev = somGetGlobalEnvironment();

// Create class object
DatastoreFactory          *pidFactDS = DatastoreNewClass (0,0);

// create objects from the class objects
Datastore          *dsm    = pidFactDS->create_object(ev);

// establish a connection
dsm->connect(ev,"DSNAME","USERID", "PASSWORD");

// or ... establish a connection as follows:
dsm->_set_datastore_name(ev,"DSNAME");
dsm->_set_user_name(ev,"USERID");
dsm->set_authentication(ev,"PASSWORD");
dsm->connect_defaults(ev);
```

Here is the C code sample:

```
#include <sdsmcon.h>

Environment *ev = somGetGlobalEnvironment();
Datastore *dsm;
dsm = DatastoreNew();

Datastore_connect(dsm, ev, "DSNAME", "USERID","PASSWORD");

/* or establish a connection as follows: */
Datastore__set_datastore_name(dsm,ev,"DSNAME");
Datastore__set_user_name(dsm,ev,"USERID");
Datastore__set_authentication(dsm,ev,"PASSWORD");
Datastore_connect_defaults(dsm,ev);
```

Accessing Database Tables

After a connection is established, you can use the generated SOM classes for accessing data in a table. There are two classes in the generated SOM IDL. One is for accessing single row and the other is for multiple rows.

The first SOM class is derived from the PersistentObject SOM class. With this class, you can add, delete, retrieve, or update a row using key.

The other SOM class allows you to retrieve multiple rows. It is a derived class of POFactory. This SOM class is identified by a suffix of "Factory" to the name of the first SOM Class. For example, if Car is the name of the first SOM class, CarFactory is the name of the other class.

You use the SOM get and set methods to retrieve and update attributes.

The following is the C++ sample code for the data access operations.

```
// create objects from the generated SOM Classes
Car            *CarPtr           = new Car;
CarFactory     *CarFactoryPtr    = new CarFactory;

// Delete a row with the key value, license = 'ABC-456'
carPtr->_set_license(ev,"ABC-456");
carPtr->del(ev);

// retrieve a row with the key value, license = '123-ZYX'
// and get the values
carPtr->_set_license(ev,"123-ZYX");
carPtr->retrieve(ev);
char CurrentModel[20] = carPtr->_get_model(ev);
char CurrentMake[20] = carPtr->_get_make(ev);

// add a row with new key value, 'MNO-456'
carPtr->_set_license(ev, "MNO-456");
carPtr->_set_model(ev, "CRX");
carPtr->_set_make(ev, "Honda");
carPtr->add(ev);

// update a row with key value, 'MNO-456'
carPtr->_set_license(ev, "MNO-456");
carPtr->_set_model(ev, "RV8");
carPtr->_set_make(ev, "MG");
carPtr->update(ev);

// retrieve a selected set of rows from the car table
_IDL_SEQUENCE_PersistentObject carList;

carList = carFactoryPtr->select(ev, "make = 'Honda'");

// use sequenceLength to get the number of selected rows
int selectedRowNumber = sequenceLength(carList);

//use sequenceElement to get a particular row (the first row)
carPtr = (Car *) sequenceElement(carList,0);

// retrieve all the rows from the table.
carList = carFactoryPtr->retrieveAll(ev);
```

Here is the C code example:

```
#include "cari.h"

Car *carPtr;
Car *carPtr1;
CarFactory *carFactoryPtr;
char CurrentModel[20];
char CurrentMake[20];
int selectedRowNumber;
_IDL_SEQUENCE_PersistentObject carList;

carPtr = CarNew();

/* Delete a row with the key value, license = 'ABC-456' */
Car__set_license(carPtr,ev,"123-ZYX");
Car_del(carPtr,ev);
```

```
/* retrieve a row with the key value, license = '123-ZYX' */
Car__set_license(carPtr,ev,"123-ZYX");
Car_retrieve(carPtr,ev);
CurrentModel = Car__get_model(carPtr,ev);
CurrentMake  = Car__get_make(carPtr,ev);

/* add a row with a new key value, 'MNO-456' */
Car__set_license(carPtr,ev,"MNO-456");
Car__set_model(carPtr,ev,"CRX");
Car__set_make(carPtr,ev,"Honda");
Car_add(carPtr,ev);

/* update a row with key value, 'MNO-456' */
Car__set_license(carPtr,ev,"MNO-456");
Car__set_model(carPtr,ev,"RV8");
Car__set_make(carPtr,ev,"MG");
Car_update(carPtr,ev);

/* retrieve a selected set of rows from the car table */
carList = CarFactory_select(carFactoryPtr, ev, "make = 'Honda'");

/* use sequenceLength to get the number of selected rows */
selectedRowNumber = sequenceLength(carList);

/* sequenceElement to get a particular row (the first row) */
carPtr2 = (Car *) sequenceElement(carList, 0);

/* retrieve all the rows from the table */
carList = CarFactory_retrieveAll(carFactoryPtr, ev);
```

Commit Work or Roll Back

Use the transact method to either commit the work you have done, or rollback the work you have done.

Here is a C++ example to commit the transaction:

```
// to commit ...
dsm.commit(ev);
```

Here is a C++ example to roll back the transaction:

```
// to rollback ...
dsm.rollback(ev);
```

Here is the C code example to commit the transaction:

```
/* code example to commit and rollback */
Datastore_commit(dsm,ev);
```

Here is the C code example to roll back the transaction:

```
/* code example to commit and rollback */
Datastore_rollback(dsm,ev);
```

Disconnecting from the Database

You must disconnect from one database before you can connect to another.

Here is an example to disconnect:

```
   // to disconnect...
   dsm.disconnect(ev);
```

Here is the C code example:

```
   /* to disconnect ... */
   Datastore_disconnect(dsm,ev);
```

Read Only Support Considerations

If the SOM classes are generated from a read only view, the update, delete, and add methods will always return an exception.

You can check the read only status by using the get method on the defaultReadOnly attribute. If the defaultReadOnly attribute is read only, the get_method returns true.

Note: The set method for this defaultReadOnly is disabled. You are not allowed to modify this attribute. This attribute is set based on the information extracted from the table (or view) in code generation.

If the defaultReadOnly attribute is not read only, the attribute returns false. If the defaultReadOnly attribute is false, you can update, delete, or add the table (or view) through the class. You can make the table or view read only by setting the currentReadOnly attribute. Once this attribute is set, the methods will always return an exception. You can check the value of the currentReadOnly using the get method.

If the defaultReadOnly attribute is true, you cannot change the value of the currentReadOnly attribute.

Here is the C++ sample code for the read only support:

```
// create car object
Car           *CarPtr          = new Car;

// Check the defaultReadOnly status before delete a row.
if (CarPtr->_get_defaultReadOnly(ev) == false) {
   // Delete a row with the key value, license = 'ABC-456'
   carPtr->_set_license(ev,"ABC-456");
   carPtr->del(ev);
}

// set the currentReadOnly to true.
CarPtr->_set_currentReadOnly(ev, true);
```

Here is the C sample code for the read only support:

```
#include "cari.h"

Car *carPtr;

/* create a car object */
carPtr = CarNew();

/* Check the defaultReadOnly status before delete a row */
if (Car__get_defaultReadOnly(carPtr,ev) == false) {
   /* Delete a row with the key value, license = 'ABC-456' */
   Car__set_license(carPtr,ev,"ABC-456");
   Car_del(carPtr,ev);
}

/* set the currentReadOnly to true */
Car__set_currentReadOnly(carPtr,ev,true);
```

Null Value Support Considerations

In the generated SOM IDL, attributes generated for each table column:

- One is for keeping the column value
- Two are for the null value support.

For example, if the table column name is called model and it is a string, the generated attributes are:

- attribute string model,
- attribute boolean modelIsNull,
- readonly attribute boolean modelIsNullable.

To check if model is a nullable attribute, you can use the get method on the modelIsNullable attribute (_get_modelIsNullable). If the result is true (nullable), you can set modelIsNull (_set_modelIsNull) to true. If the result is false (non-nullable), an exception will occur for the set method. You can also use get method to check if model is set to null or not.

Note: Once the modelIsNull attribute is set to true, the value kept in the model attribute is ignored for the data access operations. For example, the add operation will put NULL to the model column when it adds a row to the table. If the setModel method is called with a value, the NULL flag is reset.

Here is the C++ sample code for the null value support:

```
// create car object
Car           *CarPtr           = new Car;
boolean       result;

// Check if the model column is a nullable one
if (CarPtr->_get_modelIsNullable(ev) == true) {
   //model is nullable, we can set it to null
   carPtr->_set_modelIsNull(ev,true);
}

// Check if model is set to null
result = CarPtr->_get_modelIsNull(ev);
```

Here is the C sample code for the null value support:

```
#include "cari.h"

Car *carPtr;
boolean result;

/* create a car object */
carPtr = CarNew();

/* Check if the model column is a nullable one */
if (Car__get_modelIsNullable(carPtr,ev) == true) {
   /* model is nullable, we can set it to null */
   Car__set_modelIsNull(carPtr,ev,true);
} /* endif */

/* Check if model is set to null */
result = Car__get_modelIsNull(carPtr,ev);
```

Exception Handling

Exception handling used in SOM uses a separate routine to check the environment structure for exceptions occurring in the classes or in the generated files. For the SQL errors in the generated classes, you can get the sqlcode from the exception. Here is an example of the exception handling routine:

```
int exceptionCheck(void ) {
   int rc = 0;
   char *exId;
   DaxExcep_DatastoreLogonFailed erDSLogon;
   POFError *erPOF;
   POError  *erPO;

   switch (ev->_major) {
   case SYSTEM_EXCEPTION:
      cout << "system exception" << endl;
      rc ++;
      break;
   case USER_EXCEPTION:
      exId = somExceptionId(ev);
      cout << "Exception ID: " << somExceptionId(ev) << endl;
      if (strcmp(exId, ex_DaxExcep_DatastoreLogonFailed) == 0) {
         erDSLogon = (DaxExcepDatastoreLogonFailed *) somExceptionValue(ev);
         cout << "Error number: " << erDSLogon->error_number << endl;
```

```
     if (strcmp(exId, ex_POFactory_POFError) == 0) {
        erPOF = (POFError *) somExceptionValue(ev);
        cout << "Error code: " << erPOF->error_code << endl;
        if ((erPOF->error_code == DAX_REF_SQLERR) ||
            (erPOF->error_code == DAX_SEL_SQLERR)) {
          cout << "SQL code: " << erPOF->sqlcode << endl;
        }
     } else if (strcmp(exId, ex_PersistentObject_POError) == 0) {
        erPO = (POError *) somExceptionValue(ev);
        cout << "Error code: " << erPO->error_code << endl;
        if (erPO->error_code == DAX_ADD_SQLERR) {
          cout << "SQL code: " << erPO->sqlcode << endl;
        }
     }
     somExceptionFree(ev);
     rc ++;
     break;
  case NO_EXCEPTION:
     break;
  }
  return rc; }
```

The exception is checked by calling the routine after a Data Access Builder operation. Here is an example:

```
table1->update(ev);
if (exceptionCheck() !=0) {
   cout << "Update failed." << endl;
}
```

Using SOM Programs

Constructing Applications Using Data Access Builder and the Visual Builder

Data Access Builder and the Visual Builder provide a number of reusable parts to help you create your application. In addition, Data Access Builder and the Visual Builder allow you to import parts generated by the Data Access Builder into the Visual Builder This example shows you how to use these features to build two applications used for managing a car lot. These applications allows you to add and retrieve vehicle information from a database. The samples are built using Data Access, Visual Builder and WorkFrame. No coding is required.

Files to Build the Applications

To build the applications you will use a set of files specific to the applications as well as a file containing reusable parts. The files are found in the Database folder, which is in the Samples folder of the **VisualAge C++ Tools** folder.

Note: If you are not using WorkFrame, follow the directions in the README files contained in each of the directories shown below.

The application files are found in the directory :

```
\ibmcpp\samples\dax\car
\ibmcpp\samples\dax\car\DAXSAMP
\ibmcpp\samples\dax\car\CarBrws
\ibmcpp\samples\dax\car\CarEdit
```

The file *VBDAX.VBB* which contains the reusable parts is found in the directory :

```
\ibmcpp\dde4vb
```

Sample Applications Description

There are two separate applications, **CarBrws** and **CarEdit** .

The CarBrws Application

The first application, **CarBrws**, browses the contents of a car lot database. It might, for example, be used by a salesperson to help a customer identify the vehicles he would be interested in examining more closely.

The CarEdit Application

The second application, **CarEdit**, would be used to modify the contents of the database. The user can add, delete, or update the cars in the database table.

There are several windows in this application. The first window handles the database connection using a reusable visual part that is included with the Data Access Builder visual builder library.

The second window displays the database contents, and allows the user to select cars for further operations. Other windows allow the user to modify the car database.

Creating the Sample Database and the Table

The first thing you must do is recreate the sample database with the contents and tables needed for this sample.

1. Open the **DAXSAMP Database DLL Folder**. Double click on **DAXSAMP.CMD**. Enter the target drive letter for the database.

   This creates a database called *DAXSAMP* on the specified drive and invokes a command line processor script to create a table called *CAR*. The database and table take approximately 3.5 MB of disk space. The *CARV.DLL* is automatically copied to the *ibmcpp\dll* directory, and bound to the *DAXSAMP* database.

Running the Samples

Running CarBrws

1. To logon to your system, from an OS/2 command line type:

   ```
   logon userid /p:password /l:
   ```

2. Open the Data Access CarBrws Sample project. Select **Run** from the **Project** pull-down menu.
3. Choose **...or all cars** to see the complete contents of the table.
4. Choose **Only these...** to see a subset of the table based on the clause specified in the entry box.
5. Modify the clause in the entry box to specify a different filter for the car lot contents.
6. Choose **Only these...** to see the result of your modified query.
7. Close the application.

Running CarEdit

1. Open the Data Access CarEdit Sample project. Select **Run** from the **Project** pull-down menu. The connection panel will appear. The entry box for the database name will default to *DAXSAMP*.
2. Enter your userid and password. The default userid is *USERID* and the default password is *PASSWORD*.

3. Choose **Connect** to connect to the database. You can modify the name to force a connect error, displayed in a message box. If you are not or cannot be logged on, you will not be able to connect.
4. Single clicking on **"Exclusive Mode"** will ensure that this application is the only one accessing this database. Other **"Exclusive Mode"** connect attempts (for example by running **CarEdit** from a different window) will fail.
5. Changing the settings and reselecting **Connect** will cause any current connection to be disconnected, and a new connection to be made with the new parameters.
6. When a connection is successfully established, the **Disconnect**, **Commit** and **Rollback** buttons are enabled, and the second window of this application is displayed. The data is retrieved from the database and displayed in this window when the connection is made.
7. Select **Add** to add a new car. Select **Delete** to remove a car. Select **Update** to edit a car. **Commit** or **Rollback** to complete the transaction.
8. Selecting **Disconnect** will cause any active connection to be terminated. The **Disconnect**, **Commit** and **Rollback** buttons are again disabled, and the second window is hidden. **Note**: Depending on your system configuration, **Disconnect** will automatically either **Commit** or **Rollback** any pending transactions.
9. Closing the primary connection window will close the other windows and terminate the application.

Generating the Database Parts

You may skip this step. You can use the *CARV.VBE*,*CARV.DLL*, *CARV.LIB*, and *CARV.BND* files which have been provided, in which case the database access components of the sample do not need to be generated or compiled.

1. Open the DAXSAMP Database DLL Project. There are two ways to generate the database parts :
   a.
      1) Select **Database** from the **Project** pull-down menu.
      2) Choose **Create Classes** from the Startup Window.
      3) Select the DAXSAMP database, and then select **Connect**. This displays the CAR table from the DAXSAMP database.
      4) Select the USERID.CAR table, and then select **Create Classes**.
      5) Display the pop-up menu for the CAR class.

   b. Alternatively, right-click on **DAXSAMP.DAX** and select **Database**. This loads a file in which a table and class definition already exist.
2. Choose **Generate**. You can overwrite the existing files.
3. Close Data Access Builder. You do not need to save the current session when prompted.

Compiling the Database Parts

Note: You may skip this step. The *CARV.DLL*, *CARV.BND* and *CARV.LIB* files have been provided, so the database access components of the sample can be used without generation or compilation.

1. Open the DAXSAMP Database DLL project. Choose **Build** from the **Project** pull-down menu. This compiles and links the generated code. A *CARV.DLL* file and a *CARV.LIB* file are created.
2. Copy *CARV.DLL* into the directory `ibmcpp\dll`.

Building the CarBrws Application

1. Open the DAXSAMP Database DLL project. Choose **Visual** from the **Project** pull-down menu.
2. Load the file *VBDAX.VBB* (see "Files to Build the Applications" on page 255).
3. Load the file *CARBRWS.VBB*.
4. Import the file *CARV.VBE* from the directory `..\DAXSAMP`.
5. Select **CARBRWS.VBB** from the **Loaded Parts** files. Select **CARBRWS** in the **Visual Parts** list.
6. Right-click, and select **Generate->Part Source**.
7. Right-click, and select **Generate->Main for part**.
8. Close the Visual Builder
9. Select **Build** from the **Project** pull-down menu.

Building the CarEdit Application

1. Open the DAXSAMP Database DLL project. Choose **Visual** from the **Project** pull-down menu.
2. Load the file *VBDAX.VBB* (see "Files to Build the Applications" on page 255).
3. Load the file *CAREDIT.VBB*.
4. Import the file *CARV.VBE* from the directory `..\DAXSAMP`.
5. Select **CAREDIT.VBB** from the **Loaded Parts** files. Select all parts in the **Visual Parts** list.
6. Right-click, and select **Generate->Part Source**.
7. Select only *CAREDIT* from the **Visual Parts** list.
8. Right-click, and select **Generate->Main for part**.
9. Close the Visual Builder
10. Select **Build** from the **Project** pull-down menu.

22 Constructing an Application Using Data Access Builder and C++

Use Data Access Builder and the C++ classes to build an inventory application that allows you to use Data Access Builder generated classes to make adjustments to an inventory database based on changes in another database used for received orders.

The C++ Stock Sample program can be run from either:

- a project
- an OS/2 command line.

Files to Build this Application

All of the files you need to build this application are in:

```
samples\dax\stock\csetpp.
samples\dax\stock\csetpp\README.
```

Running the C++ Stock Sample from a Project

Creating the Database and the Table

1. Double click on the **DAX C++ Stock Sample Project** to open it.
2. Create the database and tables for the sample by double clicking **setupdb.cmd**. The database will be created on drive **e**. If you want to create the database on a different hard drive, click the right mouse button and **Select setting**. On the **Program** pages, change the hard drive letter for the parameters you want. The default hard drive letter is the one you created the database on.

Generating the Database Classes

1. Start Data Access Builder
2. Click mouse button 2 on **brchone.dax**. Choose **Database** on the pop-up menu. **brchone.dax** is a pre-built file containing the mappings between the database tables and the classes.

   You can create the mapping yourself, doing the following:

   a. Click mouse button 2 on any white space in the project.
   b. Choose **Database** from the pop-up menu.
   c. Choose **Create classes**. The **Create classes** window displays.
   d. Select the **BRCHONE** database and choose **Connect**. The **Tables** list box fills with two tables.

e. Choose **Select All** then choose **Create classes**. The **Create classes** window closes, and the mapping for the table icon and a class icon displays in the client area

f. Click mouse button 2 on the inventory table icon and choose **Create class** A second inventory class icon, Inventory1, displays.

g. Double click on the Inventory1 class icon to display the class notebook.

h. On the **Names** page, change the class name to `PriceList` and change the file name Invent1 to `prclist`. The mapping is complete.

i. Choose **Save** from the **File** menu.

3. Click mouse button 2 on the class icon and choose **Generate** → then choose **PARTS** from the cascade menu. There are three groups of generated files: `invento*.*`, `prclist*.*`, and `receive*.*`.
4. Close Data Access Builder.
5. Click mouse button 2 on any white space in the project. Choose **Refresh** from the pop-up menu. The generated files are shown in the project.

Compiling the Database Classes

1. Click mouse button 2 on any white space in the project. Choose **Build** from the pop-up menu to build the project.
2. Choose **Refresh** from the pop-up menu. You see **client.exe** is built.

Running the Application

1. Double click on **client.exe** to run the sample.

   The following output displays:

   The following is a price list before update:

   ```
   Prod. ID   Description            Price
   ========   ====================   ======
   RAM4-72    4M SIMM 72 pin 70 ns   $210.68
   MONAD14    ADI 14i .28 Microsc.   $409.00
   PRTZP52    ZP Inkjet 360x360      $389.00
   ```

   The following is an updated price list:

   ```
   Prod. ID   Description            Price
   ========   ====================   ======
   RAM4-72    4M SIMM 72 pin 70 ns   $296.44
   HDR0025    Hard Drive 250MD IDE   $286.36
   MONAD14    ADI 14i .28 Microsc.   $197.58
   PRTZP52    ZP Inkjet 360x360      $389.00
   HDR0034    Hard Drive 340MD IDE   $270.89
   CPU486a    486SLC-33 TI CPU,AMI   $139.99
   CPUPTMa    Pentium 60/66 PCI256   $534.34
   ```

Note: To run the application again, you must reset the database tables, double click on **resetdb.cmd**.

Constructing an Application Using Data Access Builder and SOM

Use Data Access Builder and the SOM classes to build an inventory application that allows you to use Data Access Builder generated classes to make adjustments to an inventory database based on changes in another database used for received orders.

The SOM Stock Sample program can be run from either:

- a project
- an OS/2 command line.

Files to Build this Application

All of the files you need to build this application are in:

```
samples\dax\stock\som.
samples\dax\stock\som\README.
```

Running the SOM Stock Sample from a Project

Creating the Database and the Table

1. Double click on the **DAX SOM Stock Sample Project** to open it.
2. Create the database and tables for the sample by double clicking **setupdb.cmd**. If you want to create the database on a different hard drive, click the right mouse button and **Select setting**. On the **Program** pages, change the hard drive letter for the parameters you want. The default hard drive letter is the one you created the database on.

Generating the Database Classes

1. Start Data Access Builder
2. Click mouse button 2 on **brchone.dax**. Choose **Database** on the pop-up menu. **brchone.dax** is a pre-built file containing the mappings between the database tables and the classes.

   You can create the mapping yourself, doing the following:
   a. Click mouse button 2 on any white space in the project.
   b. Choose **Database** from the pop-up menu.
   c. Choose **Create classes** from the **File** menu. The **Create classes** window displays.
   d. Select the **BRCHONE** database and choose **Connect**. The **Tables** list box fills with two tables.

   e. Choose **Select All** then choose **Create classes**. The **Create classes** window closes, and the mapping for the table icon and a class icon displays in the client area
   f. Click mouse button 2 on the inventory table icon and choose **Create class** A second inventory class icon, Inventory1, displays.
   g. Double click on the Inventory1 class icon to display the class notebook.
   h. On the **Names** page, change the class name to `PriceList` and change the file name Invent1 to `prclist`. The mapping is complete.
   i. Choose **Save** from the **File** menu.

3. Click mouse button 2 on the class icon and choose **Generate** → then choose **IDL** from the cascade menu. There are three groups of generated files: `invento*.*`, `prclist*.*`, and `receive*.*`.
4. Close Data Access Builder.
5. Click mouse button 2 on any white space in the project. Choose **Refresh** from the pop-up menu. The generated files are shown in the project.

Compiling the Database Classes

1. Click mouse button 2 on any white space in the project. Choose **Build** from the pop-up menu to build the project.
2. Choose **Refresh** from the pop-up menu. You see **client.exe** is built.

Running the Application

1. Double click on **client.exe** to run the sample.

   The following output displays:

   The following is a price list before update:

```
RAM4-72    4M SIMM 72 pin 70 ns   210.68
MONAD14    ADI 14i .28 Microsc.   409.00
PRTZP52    ZP Inkjet 360x360      389.00
```

   The following is an updated price list:

```
Prod. ID   Description            Price
========   ====================   ======
RAM4-72    4M SIMM 72 pin 70 ns   210.68
HDR0025    Hard Drive 250MD ID    239.45
MONAD14    ADI 14i .28 Microsc.   409.00
PRTZP52    ZP Inkjet 360x360      389.00
HDR0034    Hard Drive 340MD ID    270.89
CPU486a    486SLC-33 TI CPU,AM    139.99
CPUPTMa    Pentium 60/66 PCI25    459.50

Program completed successfully
```

Note: To run the application again, you must reset the database tables, double click on **resetdb.cmd**.

Part 6. The User Interface Class Library

The User Interface Class Library is one of the class libraries included in the IBM VisualAge C++ for OS/2 product. It is a C++ class library that simplifies how you develop OS/2 applications with graphical user interfaces (GUI). The User Interface Class Library provides classes that you can use in your applications and reuse to extend your applications.

You can use the User Interface Class Library classes to build applications that simulate Common User Access (CUA) workplace look and feel and take advantage of PM features. You can also use the User Interface Class Library to develop applications that are portable between the AIX and OS/2 operating systems.

24

Using the User Interface Class Library

This part enables you to start using the IBM VisualAge C++ User Interface Class Library classes and helps you learn about features that the class library provides to help you develop your own applications.

Using the User Interface Class Library, the chapter you are reading now, contains an introduction to the User Interface Class Library.

Chapter 25, “Summary of Changes” provides you with a listing of the classes and member functions that have changed in this release.

Chapter 26, “Introducing the User Interface Class Library” provides a high-level description of the User Interface Class Library. The classes are grouped into categories based on the tasks you perform when developing applications.

Chapter 27, “Creating User Interface Class Library Applications” describes the classes that make up a typical application and the classes you use to develop basic application components.

Chapter 28, “Creating and Using Windows” describes the classes that enable you to create frame windows with extensions and basic controls, as well as message boxes.

Chapter 29, “Creating and Using Text Controls” describes how you create entry fields, multiple-line edit (MLE) fields, and button controls.

Chapter 30, “Creating and Using List Controls” describes the classes that enable you to create list controls, including list boxes, combination boxes, sliders, and spin buttons.

Chapter 31, “Creating and Using Canvas Controls” describes how you create canvas controls, including split canvases, set canvases, multiple-cell canvases, and view ports.

Chapter 32, “Creating and Using File and Font Dialogs” describes the classes that enable you to create file dialogs and font dialogs.

Chapter 33, “Creating Menus” describes how you can create menus for your applications.

Using the User Interface Class Library

Chapter 34, "Creating and Using Notebooks" describes the classes that you use to create notebooks and how to add notebook styles.

Chapter 35, "Creating and Using Containers" describes how you create a container control.

Chapter 36, "Supporting Direct Manipulation" describes how to implement direct manipulation in your applications.

Chapter 37, "Defining Application Resources" describes a resource file and explains how you convert OS/2 PM resource files to AIX resource files.

Chapter 38, "Adding Events and Event Handlers" describes the classes that you use to create event handlers, mouse handlers and events, as well as explains how you can write your own handler class.

Chapter 39, "Understanding Fonts" describes how you set and change fonts.

Chapter 40, "Adding Clipboard Support" describes how to add a clipboard to your applications.

Chapter 41, "Adding Tool Bars" describes the classes that you use to create and customize tool bars for your applications.

Chapter 42, "Using Graphics in Your Application" describes the different classes provided by the User Interface Class Library to create 2-dimensional graphics.

Chapter 43, "Creating and Using Multimedia Controls" describes the multimedia classes and provides samples and instructions for their use.

Chapter 44, "Providing Help Information" explains how you add help information to your application. This section includes adding fly over help to your applications and setting time intervals.

Chapter 45, "Introducing the Sample Application" through **Chapter 51, "Adding a Font Dialog, a Pop-up Menu, and a Notebook"** take you step-by-step through the Hello World application. This application illustrates many features of the User Interface Class Library. Each version of the Hello World application builds on concepts covered in the previous versions and shows you a different aspect of the class library.

You can find sample code for the examples in the User Interface Class Library samples directory, `\ibmcpp\samples\ioc` so you can follow along and create your own examples as you read this book.

This guide also provides appendixes with commonly used information.

Appendix A, “Class Hierarchy by Category” lists all User Interface Class Library classes.

Appendix B, “New Color Support” lists all the new User Interface Class Library color support member functions.

Appendix C, “Task and Samples Cross-Reference Table” lists common tasks performed using the User Interface Class Library and the sample that demonstrates each task.

Appendix D, “Using Extended Style Support” discusses using the extended style support provided by the User Interface Class Library as solutions to a number of window style issues.

Appendix E, “Obsolete and Ignored Members Cross-Reference Tables” provides cross-reference tables listing unsupported and obsolete member functions and classes and their replacements, if any.

“Bibliography” contains lists of related books and publications.

Refer to the *IBM VisualAge C++ Open Class Library Reference* Volume II and III (hereafter called *Open Class Library Reference*) for complete reference details on the User Interface Class Library.

The Contextual Help Feature

The User Interface Class Library provides contextual help for each class and member function. To access contextual help:

- Install the CPP*.INF and CPP.NDX and CPPBRS.NDX files
- Use the VisualAge Editor

Access contextual help by positioning the cursor over the name of a class or member function in the text you are editing and pressing Ctrl-H. This opens the online version of the *Open Class Library Reference* and displays information about that class or member function.

Refer to the product installation instructions for complete details on setting the environment variables needed to use the contextual help feature.

User Interface Class Library Conventions

This section describes the User Interface Class Library conventions for the following:

- File names
- Class names and member names
- Function return types
- Function arguments
- Examples

File Names

File names have a maximum of eight characters. All files provided by the User Interface Class Library begin with the letter “I” for IBM, for example, IAPP.HPP. The following table lists file names, file extensions, and a brief description.

File Name and Extension	Description
I*xxxxxxx*.CPP	Source code
I*xxxxxxx*.H	Constant definitions file
I*xxxxxxx*.HPP	Class interface file
I*xxxxxxx*.INL	Inline functions

Refer to the *Open Class Library Reference* for cross-reference tables for the header files and the classes they contain.

The IBM VisualAge C++ product files begin with the letters “CPP”. The following table lists some file names, files extensions, and a brief description.

File Name and Extension	Description
CPPOOC3I.LIB	Import library file
CPPOOB3.DLL	Multithreaded dynamic-link library file containing the base library classes (Collection/Application Support classes)
CPPOOB3.DEF	Import module-definition file used to rebuild the CPP00B3.DLL file
CPPOOD3.DLL	Multithreaded dynamic-link library file containing the dynamic data exchange classes.
CPPOOD3.DEF	Import module-definition file used to rebuild the CPP00D3.DLL file
CPPOOM3.DLL	Multithreaded dynamic-link library file containing the Multimedia classes.
CPPOOM3.DEF	Import module-definition file used to rebuild the CPP00M3.DLL file

File Name and Extension	Description
CPPOOU3.DLL	Multithreaded dynamic-link library file containing the User Interface Base library classes.
CPPOOU3.DEF	Import module-definition file used to rebuild the CPP00U3.DLL file
CPPOOC3.LIB	Static object library
CPP.NDX	Index for online help (Does not contain class::member syntax for members)
CPPBRS.NDX	Index for online help (Contains class::member references)
CPP*.INF	Online help and online documentation files
CPPOOC3U.MSG	Exception messages

Class Names and Member Names

The following rules were used for naming the User Interface Class Library classes and members:

- Type names begin with a capital letter.
- Global type names begin with the letter “I”, as in ICurrentApplication.
- Member names, including member functions, member data, and enumerations, begin with lowercase letters, as in autoSize data member.

  Note: In this book, single-word member functions have ClassName:: added to them; for example, the member function “show” appears as IWindow::show.

Function Return Types and Function Arguments

To follow the User Interface Class Library conventions, pass objects as **const** references or references and return objects by value rather than by reference. Pass objects by pointer rather than by reference when you want a parameter to use its default.

Function return types for the various functions are as follows:

- A Boolean (true or false). Use IBase::Boolean because it is portable between OS/2 and AIX. The following is an example of a testing function:

  ```
  IBase::Boolean isValid() const;
  ```

Note: The User Interface Class Library returns a 0 if false and a nonzero if true, so do not test for the following:

```
isValid()== true
```

Instead use `if(isValid)`

- An object. Accessor functions typically return an object. An *accessor* returns information about the elements of a data type. The following example returns a pointer to an IWindow object.

  ```
  IWindow* owner();                  //Returns a pointer to an object
  ```

- An object reference. Functions that act on an object return a reference to the object on which they were called. For example:

  ```
  IWindow& hide();
  ```

  This lets you chain function calls together, as shown in the following example:

  ```
  window.moveTo(IPoint(10,10)).show();
  ```

Function arguments are passed in the following ways:

- Built-in types (integers or doubles, for example) and enumerations are passed in by value.
- Objects are passed by reference. If the argument is not modified by the function, it is passed as a **const** reference.
- Optional objects are passed by pointer. This allows a 0 pointer to signify that no object is being passed.
- IWindow objects are passed by pointer.
- IContainerObjects are passed by pointer.
- Strings are passed as a **const char ***. This enables you to pass either an IString object or an array of characters.

A Note about Samples and Examples

This part of the *User's Guide* contains both samples and examples. Samples, including the Hello World sample application, are shipped with the User Interface Class Library product in the `\ibmcpp\samples\ioc` directory. Examples, on the other hand, exist only in the *User's Guide*.

25 Summary of Changes

The User Interface Class Library version 3.0 contains the following changes and enhancements:

- New and enhanced classes
- New and enhanced functions
- New styles

For late-breaking news, refer to the README file shipped with the product.

Some header files have been optimized to include only the headers it requires. Therefore, if you obtain compiler errors stating that a class was not found, you should include the necessary header file.

New and Enhanced Classes

The following classes are new:

- Bi-directional Support

  IWindow::BidiSettings

- Control Classes

  IMouseEvent
  IMouseHandler
  IMousePointerEvent
  IMousePointerHandler

- Multimedia Classes

  IMM24FramesPerSecondTime
  IMM25FramesPerSecondTime
  IMM30FramesPerSecondTime
  IMMAmpMixer
  IMMAudioBuffer
  IMMAudioCD
  IMMAudioCDContents
  IMMCDDA
  IMMCDXA
  IMMConfigurableAudio
  IMMCuePointEvent
  IMMDevice
  IMMDeviceEvent

Classes

IMMDeviceHandler
IMMDigitalVideo
IMMErrorInfo
IMMFileMedia
IMMHourMinSecFrameTime
IMMHourMinSecTime
IMMMasterAudio
IMMMillisecondTime
IMMMinSecFrameTime
IMMNotifyEvent
IMMPassDeviceEvent
IMMPlayableDevice
IMMPlayerPanel
IMMPlayerPanelHandler
IMMPositionChangeEvent
IMMRecordable
IMMRemoveableMedia
IMMSequencer
IMMSpeed
IMMTime
IMMTrackMinSecFrameTime
IMMWaveAudio

- Notification Support Classes

  IButtonNotifyHandler
  ICircularSliderNotifyHandler
  IComboBoxNotifyHandler
  IContainerControlNotifyHandler
  IEntryFieldNotifyHandler
  IFrameWindowNotifyHandler
  IListBoxNotifyHandler
  IMenuNotifyHandler
  IMultiLineEditNotifyHandler
  INotebookNotifyHandler
  INotificationEvent
  INotifier
  INumericSpinButtonNotifyHandler
  IObserver
  IObserverList
  IScrollBarNotifyHandler
  ISettingButtonNotifyHandler
  IStandardNotifier
  ITextControlNotifyHandler
  ITextSpinButtonNotifyHandler

ITitleNotifyHandler
IWindowNotifyHandler

- Tool Bar Support Classes

  ICustomButton
  ICustomButtonDrawEvent
  ICustomButtonDrawHandler
  IToolBar
  IToolBarButton
  IToolBarContainer
  IToolBarFrameWindow

- 2-D Graphics Support Classes

  IGArc
  IGBitmap
  IGChord
  IGEllipse
  IGLine
  IGList
  IGList::Cursor
  IGPie
  IGPolygon
  IGPolyline
  IGRectangle
  IGRegion
  IGString
  IGraphic
  IGraphicBundle
  IGraphicContext
  IG3PointArc
  IPointArray
  ITransformMatrix

- Fly Over Help Classes

  IFlyOverHelpHandler
  IFlyText

- String Parsing Support Class

  IStringParser

- Timer Support Classes

  ITimer
  ITimer::Cursor
  ITimerFn

Classes

ITimerMemberFn
ITimerMemberFn0

- Spin Button Classes

  IBaseSpinButton
  INumericSpinButton
  ITextSpinButton
  ITextSpinButton::Cursor

- ListBox Classes

  IBaseListBox
  ICollectionViewListBox
  IListBoxSizeItemEvent

- ComboBox Classes

  IBaseComboBox
  ICollectionViewComboBox

- String Generation Support Classes

  IStringGenerator
  IStringGeneratorFn
  IStringGeneratorMemberFn
  IStringGeneratorRefMemberFn

- ClipBoard Support Classes

  IClipBoard
  IClipBoardHandler

- Window Coordinate Mapping Support Classes

  ICoordinateSystem

- Animated Button Support Class

  IAnimatedButton

- Additional Canvas Classes

  IDrawingCanvas

- Additional Container Class

  ICnrAllocator

- Additional Direct Manipulation Classes

  IDMMenuItem
  IDMTBarButtonItem
  IDMToolBarItem

IDMTargetEvent

- Additional Help Classes

  IHelpHyperlinkEvent

- Additional Slider Classes

  ICircularSlider
  ISliderArmHandler

New and Enhanced Member Functions

The following classes have new member functions:

Class Name	Member Function
I3StateCheckBox	convertToGUIStyle
IAccelerator	reset
IBitmapControl	convertToGUIStyle
IBuffer	includesMBCS isMBCS isValidMBCS
IButton	backgroundColor disabledForegroundColor enableNotification foregroundColor hiliteBackgroundColor hiliteForegroundColor
ICanvas	backgroundColor convertToGUIStyle
ICheckBox	convertToGUIStyle
IComboBox	convertToGUIStyle enableNotification layoutAdjustment minimumRows setMinimumRows visibleRectangle enum ControlType dropDownList
IContainerColumn	isHeadingWriteable isWriteable
IContainerObject	isWriteable operator new
ICurrentThread	isGUIInitialized initializeGUI terminateGUI

Member Functions

Class Name	Member Function
IDBCSBuffer	charLength includesMBCS isMBCS isValidMBCS isSBC maxCharLength prevCharLength
IDM	enum RenderCompletion targetSuccessful enum RenderCompletion targetFailed
IDMHandler	isContainerControl
IDMImage	defaultStyle setDefaultStyle
IDMItem	isTargetTheSource tokenForWPSObject
IDMOperation	dragWasInterrupted setDragResult setContainerRefreshOff setContainerRefreshOn
IDMSourceOperation	operation setStackingPercentage stackingPercentage
IEntryField	backgroundColor convertToGUIStyle cursorPosition enableNotification foregroundColor isWriteable removeAll selectedTextLength setChangedFlag setCursorPosition
IEvent	controlHandle controlWindow dispatchingWindow operator = setControlHandle setDispatchingHandle setHandle
IEventData	asLong
IFileDialog	convertToGUIStyle

Member Functions

Class Name	Member Function
IFileDialog::Settings	fileName initialDrive initialFileType isDialogTemplateSet isOpenDialog isPositionSet okButtonText position title
IFileDialogHandler	handleEventsFor stopHandlingEventsFor
IFont	operator = enum Direction leftToRight enum Direction topToBottom enum Direction rightToLeft enum Direction bottomToTop
IFontDialog	defaultStyle setDefaultStyle convertToGUIStyle
IFontDialog::Settings	setFont
IFrameExtension	useMinimumSize
IFrameHandler	positionExtensions
IFrameWindow	backgroundColor convertToGUIStyle disabledBackgroundColor enableNotification enum FrameSource dialogResource enum FrameSource noDialogResource enum FrameSource tryDialogResource isAnExtension mousePointer resetBackgroundColor resetDisabledBackgroundColor setLayoutDistorted setMousePointer setToolBarList toolBarList useExtensionMinimumSize
IGraphicPushButton	convertToGUIStyle marginSize setMarginSize
IGroupBox	convertToGUIStyle foregroundColor visibleRectangle
IHandle	asUnsigned
IHelpHandler	hyperlinkSelect

Member Functions

Class Name	Member Function
IIconControl	convertToGUIStyle visibleRectangle
IListBox	backgroundColor calcMinimumSize convertToGUIStyle enableNotification minimumCharacters minimumRows setItemHeight setMinimumCharacters setMinimumRows
IListBoxDrawItemEvent	isSelectionStateDrawn setSelectionStateDrawn
IListBoxDrawItemHandler	deselectItem drawItem handleEventsFor selectItem setItemSize
IMenu	add addAsNext backgroundColor convertToGUIStyle disabledBackgroundColor disabledForegroundColor enableNotification foregroundColor hiliteBackgroundColor hiliteForegroundColor isItemEnabled resetBackgroundColor resetDisabledBackgroundColor resetDisabledBackgroundColor resetForegroundColor resetHiliteBackgroundColor resetHiliteForegroundColor setForegroundColor setHiliteBackgroundColor setHiliteForegroundColor
IMenuBar	convertToGUIStyle
IMenuItem	convertToGUIStyle extendedStyle isBitmap setExtendedStyle
IMenuNotifyHandler	dispatchHandlerEvent
IMultiCellCanvas	convertToGUIStyle

Class Name	Member Function
IMultiLineEdit	backgroundColor convertToGUIStyle cursorLinePosition cursorPosition disableUpdate enableNotification enableUpdate foregroundColor isWriteable setCursorLinePosition setCursorPosition
INotebook	backgroundColor convertToGUIStyle hiliteBackgroundColor majorTabBackgroundColor majorTabForegroundColor minorTabBackgroundColor minorTabForegroundColor pageBackgroundColor resetMajorTabBackgroundColor resetMajorTabForegroundColor resetMinorTabBackgroundColor resetMinorTabForegroundColor resetPageBackgroundColor setMajorTabForegroundColor setMinorTabBackgroundColor setMinorTabForegroundColor setPageBackgroundColor setMajorTabBackgroundColor
IOutlineBox	convertToGUIStyle visibleRectangle
IPaintEvent	setGraphicContext
IProgressIndicator	backgroundColor convertToGUIStyle
IPushButton	addBorder convertToGUIStyle hasBorder removeBorder
IRadioButton	convertToGUIStyle

Member Functions

Class Name	Member Function
IRectangle	centerXCenterY centerXMaxY centerXMinY maxX maxXCenterY maxXMaxY maxXMinY maxY minX minXCenterY minXMaxY minXMinY minY
IResourceLibrary	tryToLoadBitmap tryToLoadIcon tryToLoadMessage tryToLoadString
ISWP	isHide isMove isShow isSize isZOrder setHide setMove setNoAdjust setShow setSizeFlag setZOrder
IScrollBar	convertToGUIStyle enableNotification foregroundColor hiliteForegroundColor
ISetCanvas	setGroupPad convertToGUIStyle groupPad
ISettingButton	enableNotification
ISlider	convertToGUIStyle
ISplitCanvas	convertToGUIStyle resetSplitBarEdgeColor resetSplitBarMiddleColor setSplitBarEdgeColor setSplitBarMiddleColor splitBarEdgeColor splitBarMiddleColor visibleRectangle

Member Functions

Class Name	Member Function
IStaticText	backgroundColor convertToGUIStyle fillColor foregroundColor resetFillColor setFillColor
IString	includesMBCS isMBCS isValidMBCS
IThread	autoInitGUI messageQueue variable setAutoInitGUI setVariable setDefaultAutoInitGUI
ITitle	activeColor activeTextBackgroundColor activeTextForegroundColor borderColor inactiveColor inactiveTextBackgroundColor inactiveTextForegroundColor enableNotification resetActiveTextBackgroundColor resetActiveTextForegroundColor resetInactiveTextBackgroundColor resetInactiveTextForegroundColor setActiveTextBackgroundColor setActiveTextForegroundColor setInactiveTextBackgroundColor setInactiveTextForegroundColor setViewNumber viewNumber
IViewPort	convertToGUIStyle

Member Functions

Class Name	Member Function
IWindow	activeColor addObserver applyBidiSettings backgroundColor borderColor capturePointer characterSize convertToGUIStyle disableNotification disabledBackgroundColor disabledForegroundColor dispatchRemainingHandlers enableNotification enum BidiLayout enum BidiNumeralType enum BidiTextOrientation enum BidiTextShape enum BidiTextType enum EventType control extendedStyle font foregroundColor handleWithParent handleWithPointerCaptured hasPointerCaptured hiliteBackgroundColor hiliteForegroundColor inactiveColor isBidiSupported isEnabled isEnabledForNotification isGroup isTabStop isWindowValid movePointerTo nativeRect notificationHandler notifyObservers observerList parentSize pointerPosition

Class Name	Member Function
IWindow cont.	releasePointer removeAllObservers removeObserver resetActiveColor resetBackgroundColor resetBorderColor resetColor resetDisabledBackgroundColor resetDisabledForegroundColor resetFont resetForegroundColor resetHiliteBackgroundColor resetHiliteForegroundColor resetInactiveColor resetShadowColor setActiveColor setBackgroundColor setBorderColor setDisabledBackgroundColor setDisabledForegroundColor setExtendedStyle setFont setForegroundColor setHiliteBackgroundColor setHiliteForegroundColor setId setInactiveColor setNotificationHandler setShadowColor shadowColor windowWithOwner windowWithParent

Enhanced Member Functions

The following classes have changed member functions:

Class Name	Member Function
IApplication	adjustPriority setPriority
IButton	highlight IButton
ICanvas	ICanvas
ICnrDrawHandler	ICnrDrawHandler
ICnrMenuHandler	ICnrMenuHandler
IColor	IColor

Member Functions

Class Name	Member Function
IComboBox	add setItemText remove
IControl	IControl
ICurrentThread	initializePM
IDDEClientConversation	acknowledged beginHotLink conversationEnded data endHotLink executeAcknowledged hotLinkInform pokeAcknowledged pokeData requestData
IDDETopicServer	acceptConversation acknowledged beginHotLink conversationEnded executeCommands hotLinkEnded pokeData requestData requestHotLinkData
IDMHandler	setItemProvider enableDragDropFor enableDragFrom enableDropOn
IDMImage	IDMImage style
IDMSourceHandler	sourceBegin
IDMTargetHandler	targetDrop
IDMTargetRenderer	informSourceOfCompletion
IEntryField	clear copy cut selectRange
IEvent	setResult
IEventData	operator char* operator unsigned long
IFileDialog	IFileDialog

Member Functions

Class Name	Member Function
IFont	setCharHeight setCharSize setCharWidth setFontAngle setFontShear setName setPointSize useBitmapOnly useNonPropOnly useVectorOnly
IFontDialog	IFileDialog
IFontDialogHandler	modelessResults
IFrameExtension	separator setSize width
IFrameWindow	addExtension borderHeight borderWidth IFrameWindow notifyOwner removeExtension setBorderHeight setBorderSize setBorderWidth setExtensionSize tryToLoadDialog
IHelpHandler	controlSelect handleError helpUndefined keysHelpId menuBarCommand openLibrary showContents showCoverPage showHistory showIndex showPage showSearchList showTutorial subitemNotFound swapPage
IHandle	operator Value
IHelpNotifyEvent	controlId
IIconControl	IIconControl

Member Functions

Class Name	Member Function
IListBox	add locateText remove setHeight setItemText
IMenu	IMenu
IMenuItem	attribute index isSubmenu isText setAttribute setIndex setStyle style
IMultiLineEdit	clear copy cut selectRange setChangedFlag
IPaintHandler	IPaintHandler
IPrivateResource	handle
IProgressIndicator	setHomePosition setPrimaryScale
IResizeHandler	IResizeHandler
IResource	handle
IResourceId	IResourceId
ISelectHandler	ISelectHandler
ISettingButton	select
ISharedResource	handle
IShowListHandler	IShowListHandler
ISize	asSIZEL
IStaticText	setAlignment
ISubmenu	addBitmap addItem addSeparator addText
ISWP	ISWP
ITextControl	ITextControl
IThread	setAutoInitPM setDefaultAutoInitPM

Member Functions

Class Name	Member Function
IWindow	windowWithHandle windowWithId

Additional Library Enhancements

Existing member functions have been updated to conform to the following User Interface Class Library conventions:

- Many functions are now declared as "const".
- Many parameters are now passed by "const" references instead of being passed by value.
- Many functions are now declared as "virtual".

Many of the include statements in the header files have been replaced with forward declares. This was done so that applications will only have required files included. You might need to add include statements in your application to replace the removed includes.

In place of FID_CLIENT and 0x8008 for the window identifier of a client window, you can now use the #define IC_FRAME_CLIENT_ID.

Window identifiers of the child windows created by an IViewPort have been changed. The changes are:

Child Window	Old Window id	New Window id
Vertical Scroll Bar	0x8000	IC_VIEWPORT_VERTSCROLLBAR
Horizontal Scroll Bar	0x8001	IC_VIEWPORT_HORZSCROLLBAR
View Rectangle	0x8002	IC_VIEWPORT_VIEWRECTANGLE

IWindow::clipToParent and IWindow::synchPaint have been removed from the default style of ICanvas.

INotebook::setStatusText(const char*) now throws an IInvalidRequest exception when invoked and the page does not allow status text.

IEntryField::selectedRange() and IMulitLineEdit::selectedRange() throws an IInvalidRequest when invoked and there is no text selected.

A message box will always have a title bar when IMessage::moveable style is specified, even if there is no title text.

New Styles

The following classes have new styles:

Class Name	New Style
IComboBox	readOnlyDropDownType
IContainerControl	noSharedObjects
ISetCanvas	decksByGroup

Extended Styles Support

Extended styles are now supported with the ICLUI framework. New functions have been added to the IWindow hierarchy. See Appendix D, "Using Extended Style Support" on page 677 for more details.

26 Introducing the User Interface Class Library

The User Interface Class Library contains over 400 classes and over 4000 member functions. To assist you in learning about the classes and to guide you as you start developing applications, we organized the classes into the following basic categories:

- Application Control Classes
- Data Types and Base Classes
- Base Windows, Menus, Handlers, and Events
- Basic Controls
- Advanced Controls, Dialogs, and their Handlers
- Direct Manipulation Classes
- Dynamic Data Exchange Classes
- 2-Dimensional Graphic Classes
- Multimedia Classes

Application Control Classes: Provide support for the application, threads, timers, profiles, and the resources used by the applications you develop.

Data Types and Base Classes: Provide support for the exceptions, trace output, messages, strings, notifications, and window geometry used by the applications you develop.

Model basic data types, such as strings, points, and rectangles. These classes hide the structure of the data, while providing the capability to access and alter the data. In addition, a set of handle classes are provided to access window or application-specific handles.

Base Windows, Menus, Handlers, and Events: Provide support for the basic windows, handlers, events, and menus used by the applications you develop.

Encapsulates the basic graphical building blocks that are used to construct application windows. This allows window position and appearance (parent windows) to be separated from event handling (owner windows).

Encapsulate the user's interaction with application windows. The library creates event objects as a result of some action by the user or by other applications. These event objects contain information about what occurred; they are passed to handler objects for processing. Each window has some default event processing; however, the application can create instances of the handler classes to process certain event objects to override the default behavior.

Use handlers to override or augment a control's default behavior. For example, use a handler when you want to take some action based on user input such as selecting a button or list box item.

Basic Controls: Provide support for the basic controls like entry field, static text and buttons used by the applications you develop.

Advanced Controls, Dialogs, and their Handlers: Provide support for the advanced controls like container, notebook, tool bar, and the font and file dialogs used by the applications you develop.

Direct Manipulation Classes: Provide support for the direct manipulation used by the applications you develop.

Dynamic Data Exchange Classes: Provide support for the Dynamic Data Exchange (DDE) used by the applications you develop.

2-Dimensional Graphic Classes: Provide support for the 2D graphic elements used by the applications you develop.

Multimedia Classes: Provide support for the multimedia devices and controls used by the applications you develop.

Creating Your Own Classes

Most applications require new classes, which you can derive from existing classes. You derive new classes for two reasons:

- To inherit implementation details from a base class
- To substitute for a base class

The following table provides a starting point to determine which base class to use:

Added New Function	**Derived From**
Attribute	IBase or IVBase
Canvas class	ICanvas
Control	IControl or ITextControl
Cursor	IVBase
Data type	IBase or IVBase
Dialog window	IFrameWindow
Event	IEvent

Added New Function	Derived From
Primary or secondary window	IFrameWindow
Settings	IBase
Style	IBitFlag
Window behavior	Handler specific to the window

Understanding the Design Recommendations

This section gives recommendations for designing User Interface Class Library applications. These general recommendations should not substitute for detailed design guidelines. Many of the topics listed here require a great deal of consideration when you design complex object-oriented applications.

The Hello World sample application shipped with this product uses these design recommendations.

Reviewing C++ Recommendations

The following topics are general C++ recommendations.

Choosing Multiple Inheritance or Composition

It is easier to inherit from multiple classes when you design simple applications. Because all of the functions from the derived classes are immediately available, you can easily use them as-is and not override them.

However, as your application evolves into a more complex application, it can be difficult to anticipate how changes in the functions of the inherited classes will affect the derived class.

Generally, if the class you design *is-a*, for example, frame window, then it should inherit from the IFrameWindow class. Inheriting from IFrameWindow is typical. However, if the class *has-a*, for example, command handler, ICommandHandler should be represented by a member in the derived frame window. It should not inherit from the command handler class. The Hello World version 3 sample application provides an example.

Refer to Chapter 48, "Adding a Command Handler and Menu Bars" on page 613 for more information about this sample.

Overriding Virtual Functions

When you override inherited member functions, such as the ICommandHandler::command function, that are defined as virtual, you should declare the overriding function as virtual too. This improves the readability of the inheriting class by saving the reader from having to search up the inheritance chain to discover that the function was originally defined as virtual.

Deleting Objects Created with New

If you create objects dynamically by using the new operator, you should delete them by using the delete operator. If an object is composed of dynamically created objects, that is, you create the composed objects with the new operator in the constructor of the composing object, then you should delete the object in the destructor of the composing object.

Note: An exception to this rule is when you use the autoDelete behavior of IWindow derived classes. Refer to IWindow::setAutoDeleteObject in the *Open Class Library Reference* for more information.

If you want to create objects that exist for the lifetime of the main window object, do not create them using new. Instead, create them as static objects (for example, in the frame window constructor).

Understanding the User Interface Class Library Recommendations

The following topics are general User Interface Class Library recommendations.

Using String Resources

The values of strings in applications vary by user because of preference or national language, for example. Therefore, you should define strings outside the application. In OS/2 this capability is provided by using OS/2 Presentation Manager (PM) resource compiler (.RC) files. This format lets you use descriptive tags to identify tables of strings and associate them with unique IDs in your application.

Motif In AIX, you can also use the .rc files, as long as you convert the files using the ipmrc2X tool. The User Interface Class Library provides the ipmrc2X tool to convert the PM .RC format to X Toolkit resource files. For more information about ipmrc2X, refer to “Converting Resource Files” on page 463. Hello World version 2 shows how to use User Interface Class Library functions to reference strings from resource files.

Refer to Chapter 47, “Adding a Resource File and Frame Extensions” on page 603 for more information about this sample.

Defining Menus in Resource Files

For ease of programming and to accommodate user preference and national languages, the User Interface Class Library provides the ipmrc2X tool to support defining menu bars, submenus, menu items, and accelerators in .RC file format. By using the same ID for the menu bar and the frame window, you can define the layout, menu item text, and accelerator key definitions external to application logic. Hello World version 3 or version 4 demonstrates this feature.

Refer to Chapter 48, “Adding a Command Handler and Menu Bars” on page 613 or Chapter 49, “Adding Dialogs and Push Buttons” on page 621 for more information about these samples.

Using Canvases Instead of Dialog Templates

OS/2 PM provides support for dialog templates, which are “layouts” of frame windows and controls. This support is not available in Motif and, therefore, is not portable. Instead, use the canvas classes, such as IMultiCellCanvas and ISetCanvas, for designing portable dialogs across AIX and OS/2. Hello World version 4 demonstrates this feature.

Refer to Chapter 49, “Adding Dialogs and Push Buttons” on page 621 for more information about this sample.

Defining the Client Window ID

In OS/2 PM, applications that define client windows should use the window ID 0x8008 because IFrameWindow::setClient changes the ID to that value regardless of the ID passed to it. Therefore, if you write portable applications, use 0x8008 as the ID for all client windows on both OS/2 and AIX. The User Interface Class Library also provides a `#define IC_FRAME_CLIENT_ID` statement for the 0x8008 ID.

Design Recommendations

27 Creating User Interface Class Library Applications

To create a User Interface Class Library application, you need to know which files to create and what goes into them. The following list describes the minimum files required for an application. Typically, the name of each file is the same; only the extensions differ.

File Name	Contains
filename.CPP	Primary C++ code for your application.
filename.HPP	Declaration of any class or classes that you create. You can put each class in a separate .HPP file or all classes in one file. If your classes are used in only one .CPP file, they can be declared in that .CPP file instead.

As you create more complex applications, separate your code into different files. The following files are optional:

File Name	Contains
filename.RC	Application resource file and associated resources used when the application requires data, such as text strings or bit maps, from an external source. Examples of external sources include .BMP, .ICO, and .DLG files.
filename.H	Header file, which defines constants used in a resource (.RC) file.
filename.DEF	Module definition file, which holds information that defines your application for the linker.
filename.IPF	Text and tags to produce the help information for your application.
filename.MAK	makefile, which holds information to compile and link your application.

When you use the User Interface Class Library to write applications, use the following structure for your files:

- #include statements

  Insert #include statements at the beginning of the file to specify other files that contain information that your application requires. The following order is recommended for #include statements in an application:

  1. Standard C library headers
  2. OS/2 Toolkit headers
  3. User Interface Class Library headers
  4. Your class headers

  Note: In certain cases, the User Interface Class Library headers can detect whether the OS/2 Toolkit headers are included and can define some Toolkit-specific functions.

  Typical #include statements are:

  - #include <Ixxxxx.HPP>

    Includes the header file that contains information about a User Interface Class Library class that your application uses. You must include the header file for each class you use. All User Interface Class Library header files begin with the letter “I”.

    Refer to the *Open Class Library Reference* for cross-reference tables for header files and the classes they contain.

  - #include "xxxxx.HPP"

    Represents the inclusion of a header file that contains the definition of a class that you created. Include header files for classes that you create if your source file uses those classes. See “Creating Your Own Classes” on page 290 for more information.

  - #include "xxxxx.H"

    Includes the file that defines your symbolic definitions.

- The function called “main” defines the application’s entry point

  Create the primary application window, call its functions to change settings, such as color, size, or position; call its inherited frame window functions to give it focus and have it displayed; and call ICurrentApplication::run to begin event processing.

- Constructor for the application window

  Typically, a new class derived from IFrameWindow defines the main application window. This new class’ constructor can follow the Main function. Its purpose is to initialize the inherited IFrameWindow, initialize data members that compose the new class, start event handlers, and set controls used in the frame window.

  Once the application window is constructed, your application can call other classes to insert controls and dialogs into the window and to handle events.

- Destructor

  Use a destructor to stop event handlers and delete any composed objects that you created using the new operator, except those IWindow objects set as autoDelete.

- Member functions for the application window

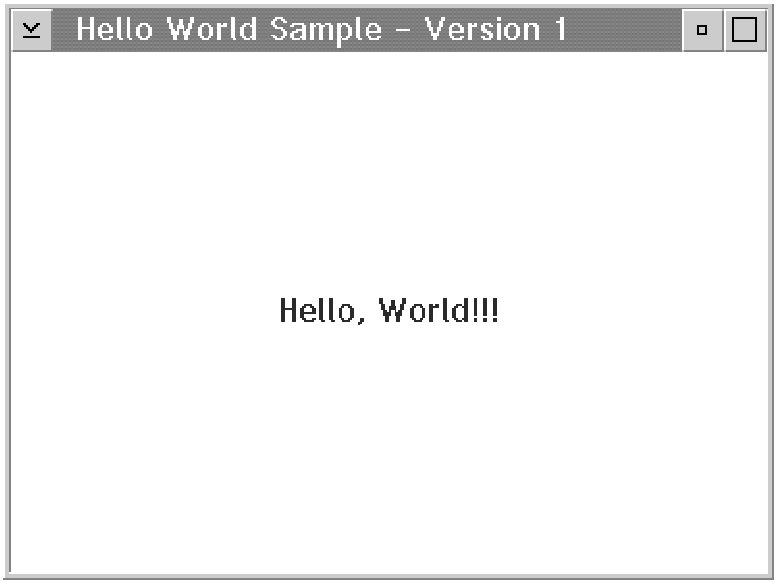
If the new class that defines the application window defines any new functions or overrides inherited functions, include them here.

Understanding a User Interface Class Library Application

An easy way to understand how the classes and objects work together is to look at a simple application, called Hello World version 1. This application has two basic user interface components:

- A standard frame window with a title bar, system menu, border, and minimize and maximize buttons.
- The rest of the window, called the client area, that contains the phrase "Hello World!!!"

The main window for Hello World version 1 looks like this:



Hello, World!!!

Figure 24. Hello World Version 1 Main Window

One source file, the .CPP file, is required for this application.

Creating a C++ Source File

The Hello World application, versions 1 through 6, illustrates many User Interface Class Library features. Hello World version 1 has only a .cpp file. This file is the C++ source file used by the C++ compiler to generate the executable part of this application. A copy of the “Hello World” version 1 application is in the `\ibmcpp\samples\ioc\hello1` directory.

The listing of the C++ source file for the Hello World version 1 application follows:

```
1  //Include User Interface Class Library class headers:
2  #include <iapp.hpp>                     //IApplication class
3  #include <istattxt.hpp>                 //IStaticText class
4  #include <iframe.hpp>                   //IFrameWindow class
5
6  /**************************************************************************
7  * main  - Application entry point for Hello World Version 1.              *
8  *         This simple application does the following:                     *
9  *           1) Creates a new object mainWindow of class IFrameWindow      *
10 *           2) Creates a new object hello of class IStaticText            *
11 *           3) Sets the static text value and aligns it                   *
12 *           4) Sets the static text as the client of the mainWindow       *
13 *           5) Sets the size of mainWindow                                *
14 *           6) Sets the window focus to mainWindow                        *
15 *           7) Displays the mainWindow                                    *
16 *           8) Starts the events processing for the application           *
17 **************************************************************************/
18 int main()
19 {
20   IFrameWindow mainWindow ("Hello World Sample - Version 1", 0x1000);
21
22   IStaticText hello(IC_FRAME_CLIENT_ID, &mainWindow, &mainWindow);
23   hello.setText("Hello, World!!!");
24   hello.setAlignment(IStaticText::centerCenter);
25   mainWindow.setClient(&hello);
26
27   mainWindow.sizeTo(ISize(400,300));
28   mainWindow.setFocus();
29   mainWindow.show();
30   IApplication::current().run();
31   return 0;
32 } /* end main */
```

This application creates the following objects:

mainWindow This IFrameWindow object is the main window for the application. It is constructed in line 20.

hello This is the static text control (IStaticText) object that contains the phrase “Hello World!!!” This object is constructed on line 22.

ISize object A temporary ISize object is created on line 27.

Starting Event Processing

To develop a User Interface Class Library application, you can use IApplication and the single instance of its derived class, ICurrentApplication. Objects of the ICurrentApplication class represent the application that is currently running.

Event Processing

To start event processing for a C++ application using the User Interface Class Library, obtain a reference to ICurrentApplication by using the static member function IApplication::current. The instance of this class contains information about the application that is accessible to the ICurrentApplication::run member function executing in the process. The following example comes from the Hello World version 1 source file:

```
⋮
30   IApplication::current().run();
⋮
```

Loading Resources into an Application

There are two ways to load resources into your application:

1. Code the values directly in the file
2. Create a resource file

The User Interface Class Library loads resources where necessary for an application. Use the ICurrentApplication member function setUserResourceLibrary to identify which resource library will be used if none is specified on a call that loads a resource. The following highlighted code shows an example.

```
int main(int argc, char **argv)             //Main function with arguments
 {
  IApplication::current().                  //Save the command line arguments
    setArgs(argc, argv);                    //in the current application object.

  IString Dllname(IApplication::current().argv(1));

  IApplication::current().                   //Get current application
   setUserResourceLibrary(dllname.asString()); //  Set the name of resource DLL.

  AHelloWindow mainWindow (WND_MAIN);       //Create main window

  IApplication::current().run();            //Get current & run the application
  return 0;
} /* end main */
```

First, the library tries to create a frame window by loading a dialog with the WND_MAIN ID from specified resource library.

You can also determine the current user resource library by calling the userResourceLibrary member function. The following highlighted code shows an example.

```
AHelloWindow :: AHelloWindow(unsigned long windowId)
  : IFrameWindow(windowId)            //Call IFrameWindow constructor
{
  hello = new IStaticText(WND_HELLO,   //Create static text control
    this, this);                       //  Pass in myself as owner & parent
  hello->setText(
   IApplication::current().
   userResourceLibrary().asString());
```

```
                                       //Set text in static text control
  hello->setAlignment(                 //Set alignment to center in both
   IStaticText::centerCenter);         //  directions
  setClient(hello);                    //Set hello control as client window

  setFocus();                          //Set focus to main window
  show();                              //Set to show main window
} /* end AHelloWindow :: AHelloWindow(...) */
```

Recording and Querying Command Line Arguments

With ICurrentApplication, you can record and query the command line arguments of your application. Set the arguments by calling setArgs with the arguments that were passed to the main function.

To query the number of arguments, use the member function ICurrentApplication::argc. This member function always returns a nonzero value because it has at least the name of the application as a parameter.

To get the nth parameter, use the member function ICurrentApplication::argv, where the argv(0) component is always the name of the application. Because argv is returned as an IString, you can use all the overloaded operators for this class.

For example, the following highlighted (bold face) code records the command line parameters.

```
int main(int argc, char **argv)
{
  IApplication::current().setArgs(argc, argv);
⋮
```

where

argc is the number of arguments received

argv represents the actual arguments.

Refer to the *Open Class Library Reference* for more information about the ICurrentApplication and IApplication classes.

Compiling and Linking Your User Interface Class Library Application

The make utility helps you manage your software development projects and the files associated with your project. Make uses a *makefile* to convert your source code file into an object file. The makefile is a special file containing a list of tasks that you provide to convert your source file.

The following example shows the Hello World version 1 makefile for the OS/2 operating system.

```
# Make file assumptions:
#    - Environment variable INCLUDE contains paths to:
#       VisualAge C++ target_directory\include;
#       OS/2 Developer's Toolkit target_directory include paths
#    - Environment variable LIB contains paths to:
#       VisualAge C++ target_directory\lib;
#       OS/2 Developer's Toolkit target_directory lib paths
#    - Current directory contains source files.  Originals are in:
#       VisualAge C++ target_directory\samples\ioc\hello1
#    - current directory will be used to store:
#       object, executable, and resource files

ERASE=ERASE
GCPPC=ICC.EXE
GLINK=ICC.EXE

ICLCPPOPTS=/GM /GD

GCPPFLAGS=$(LOCALOPTS) $(ICLCPPOPTS)

GCPPLFLAGS=/B" /pmtype:pm"

all:  HELLO1.EXE


HELLO1.EXE:  AHELLOW1.OBJ
        $(GLINK) $(GCPPLFLAGS) /Fe"HELLO1.EXE" $(CPPOLIBS) \
          AHELLOW1.OBJ

AHELLOW1.OBJ:  AHELLOW1.CPP
        $(GCPPC) /C $(GCPPFLAGS) AHELLOW1.CPP

clean:
        $(ERASE) HELLO1.EXE
        $(ERASE) AHELLOW1.OBJ
```

There are two ways to invoke make, depending on what you name your make file.

1. If you have a file named “makefile”, type the following:

   ```
   nmake
   ```

2. If you have a file by a different name, for example, makefile.OS2, type the following:

   ```
   nmake -f makefile.OS2
   ```

A successful make generates the files you need for linking and executing. To remove the files you generate, for example if you want to put the sample directory back the way you found it, run the `make clean` command. The makefiles provided with Hello World samples include a clean section that erases files you generate. The following make clean command is set at the beginning of the Hello World makefiles, to erase the files:

```
clean:
       $(ERASE) HELLO1.EXE
       $(ERASE) AHELLOW1.OBJ
```

Using the Conversion Tools

OS/2 resource files and bit-maps, and icons cannot be directly used in Motif. C Set ++ for AIX provides a set of utilities which convert resource files to X resource files and bit-maps and icons to .xpm files. The ipmrc2X tool converts OS/2 resource files into X resource files.

Note: Also, you should recompile all OS/2 .IPF (help) files that you want to use on AIX.

Linking an Application to the Open Class Library

Link your application to the Open Class Library in one of the following ways, depending on your needs:

1. If you are developing an application, use the CPPOOC3I.LIB import library.

   Your application links dynamically to the following .DLLs at run time:

   - CPPOOB3.DLL
   - CPPOOD3.DLL
   - CPPOOM3.DLL
   - CPPOOU3.DLL

   Using these .DLLs during development reduces the time it takes to link your application and the amount of swap space used.

   You must also link with CPPOM30I.LIB. This library resolves C++ runtime external references and uses CPPOM30.DLL.

   By compiling with the `-Gd -Gm` options, the compiler will automatically generate the appropriate .LIB names in the resulting .OBJ files. The linker will then use these files without you explicitly listing them. Compiling with `-Gn` or linking with `/NOD` will suppress the use ot the compiler generated .LIB names. You may need to do this if you rebuild the DLLs.

2. If you are delivering an application, you must decide whether to rename the Open Class Library .DLLs and ship them with your application or statically link the Open Class Library code directly into your executable. To statically link to the Open Class Library code, use the CPPOOC3.LIB static object library by coding the `-Gd- -Gm` compiler options.

   Using this library does not create dependencies on the four DLLs. You must also link your application to CPPOM30.LIB which resolves C++ runtime external references.

Note: You must link your application using ICC.EXE with the -Tdp compiler option.

By compiling with the appropriate +Gd option, the compiler will automatically generate the appropriate .LIB names in the resulting .OBJ files. The linker will then use these files without you explicitly listing them.

The following additional rules apply when you build your application with the dynamic libraries, instead of the static object libraries:

1. A .DLL using the Open Class Library must link dynamically to the Open Class Library code (that is, you must link with CPPOO3CI.LIB).
2. An .EXE using the Open Class Library and calling a .DLL that also uses the class library must link dynamically to the Open Class Library (that is, you must link with CPPOO3CI.LIB).
3. An .EXE or .DLL file should not link both dynamically and statically to the User Interface Class Library code.

Rebuilding the Open Class Library DLLs

If you deliver a renamed version of the Open Class Library .DLLs with your application, you can reduce the size of the .DLLs by rebuilding them and leaving out the classes that your application does not use.

A smaller .DLL takes up less space on your installation media and can also result in faster load time for the applications that use the .DLL.

How to Rebuild

Use the following steps to rebuild the .DLLs:

1. Make \ibmcpp\iocdll your current directory.

   If VisualAge C++ is installed in \ibmcpp, type:

   ```
   cd \ibmcpp\iocdll
   ```

   Note: These instructions assume VisualAge C++ is installed in \ibmcpp.

2. Extract the needed .OBJ files from the Open Class Library static library.

   To get the best performance for your rebuilt .DLLs, you must link the object files in the order specified in the .RSP files. To do this, extract the .OBJ files from the Open Class Library static libraries instead of relinking the DLL by using the static libraries directly.

   To extract the needed .OBJ files and put them in the \ibmcpp\iocdll directory, type:

   ```
   GETOBJS ..\LIB\CPPOOC3.LIB
   ```

3. Modify the .DEF and .RSP files.

Rebuilding the DLLs

To remove a class from your rebuilt .DLL, you must first determine the name of the .OBJ file in which the class implementation resides. However, be aware that some .OBJ files contain more than one class implementation. If your application uses *any* of the classes that an .OBJ file implements, you cannot remove it.

The cross-reference tables in the appendixes of the *Open Class Library Reference* can help you determine the .OBJ file that implements a given class. Although this table lists the .HPP file, you can generally substitute .OBJ for .HPP to determine the right name for the .OBJ file.

For some classes, such as IString and IResourceLibrary, a single .HPP file declares the class, but multiple .OBJ files contain the implementation. In these cases, a number is appended to the file name to make the .OBJ file names unique. For example, the implementations of the classes declared in IRESLIB.HPP are in IRESLIB.OBJ, IRESLIB1.OBJ, IRESLIB2.OBJ, IRESLIB3.OBJ, and IRESLIB4.OBJ. For sequentially numbered .OBJ files such as these, either remove all of the .OBJ files or do not remove any.

Once you determine the .OBJ files that you do not need for your rebuilt DLL, do the following:

- Edit the .RSP files and remove the lines that list the unneeded .OBJ files.
- Edit the .DEF files and remove all lines that export the functions contained in the .OBJ files whose lines you removed from the .RSP files.

The .RSP and .DEF files are found in the `\ibmcpp\iocdll` directory.

For example, if you do not use the ITime class, delete the lines that export the ITime functions. The export statements for the ITime class follow the comment that identifies the ITIME.OBJ file. For example:

```
;
; --> Object: C:\DRVRGM3\IBASE\OBJ\ITIME.obj
;
__ls__FR7ostreamRC5ITime  @656 noname        ----> DELETE
__ct__5ITimeFRC6_CTIME  @657 noname          ----> DELETE
asCTIME__5ITimeCFv  @658 noname              ----> DELETE
asString__5ITimeCFPCc  @659 noname           ----> DELETE
;
```

The following table can help you identify groups of files that you can delete. This table provides the file name pattern and the conditions under which you can delete all files that match the pattern:

File Name Pattern	Can Be Deleted if Your Application...
N*.OBJ	Never uses I_NO_INLINES when compiling
ICNR*.OBJ	Does not use container controls

File Name Pattern	Can Be Deleted if Your Application...
IDDE*.OBJ	Does not use dynamic data exchange (you do not need CPPOOD3.DLL)
IDM*.OBJ	Does not use direct manipulation or toolbar.
IMM*.OBJ	Does not use multimedia (you do not need CPPOOM3.DLL)

4. Delete unneeded .OBJ files from the `\ibmcpp\iocdll` directory.

   You can delete the .OBJ files in the current directory once you are sure you no longer need them for rebuilding. If you do this, do *not* delete the DDE4UDLL.OBJ file.

   We ship this file in the `\ibmcpp\iocdll` directory because it is not available from any of the static libraries. You need this file if you ever attempt to rebuild. (It is the .DLL Init/Term routine.) You may want to put a backup copy of this file in another directory.

   A safe way to delete the unneeded files is to type the following commands:

   ```
   DELETE I*.OBJ
   DELETE N*.OBJ
   ```

Reserved Pragma Priority Values

The User Interface Class Library reserves the use of #pragma priority values in the range of -2147482624 through -2147481600. The C++ compiler reserves the range below that. As a result, avoid using a #pragma priority value less than -2147481599 (this is equivalent to INT_MIN + 2048) to control the order of static object construction in your User Interface Class Library application.

🕮 See the *VisualAge C++: Language Reference* for more information on #pragma priority values.

Pragma Priority

28 Creating and Using Windows

When you develop an application, you usually start with a window that is a composite of a frame window, title bar, menu bar, system menu, maximize and minimize buttons, client window, and extensions. The frame window coordinates the actions of the frame extensions and client window, enabling the composite window to act as a single unit. A frame's client window is the control that corresponds to the rectangular portion of the frame window not occupied by other controls (menus or buttons).

The User Interface Class Library provides classes that construct the frame window and that let you add a variety of styles and controls.

Creating a Frame Window

A *frame window* is a window that an application uses as the base when constructing a main window or other composite window, such as a dialog window or message box. A frame window provides basic features, such as borders and a title bar. It can also have a set of resources associated with it, such as icons, menus, and accelerators.

The *parent* window is used mainly for window location. For example, most frame windows have the desktop as a parent so that they can appear anywhere on the desktop. A *child* window is a window that is only visible inside of another window, the parent. The child is "clipped" to its parent, meaning it is confined within it, and it cannot move beyond the boundaries of the parent window.

The *owner* window is mainly used for message passing. Several messages are passed up the owner chain for processing and changes to color and font are passed down the owner chain. When you move an owner window, all of the windows that it owns are moved as well except if the IFrameWindow::noMoveWithOwner style is specified. A parent window can be an owner window.

A *primary* window has the desktop as its parent and owner. A *secondary* window has the desktop as its parent, but its owner is another window. A child window would have another window for its parent and owner.

Note: A frame window can be a primary window, secondary window, or a child window.

Use the IFrameWindow class to create a frame window. The default style of the IFrameWindow class has a title bar, system menu, minimize button, maximize button, and border. The default style adds an entry for the frame window to the system window list.

The IFrameWindow class also provides several other styles. You can, for example, associate an accelerator key table to the frame window or provide an icon to be used when the window is minimized.

When you construct an IFrameWindow with a style of IFrameWindow::minimizedIcon, IFrameWindow::accelerator, or IFrameWindow::menuBar, resources corresponding to the style must be in the resource library you use to construct the frame. This library is usually the default user library, which you may use by typing:

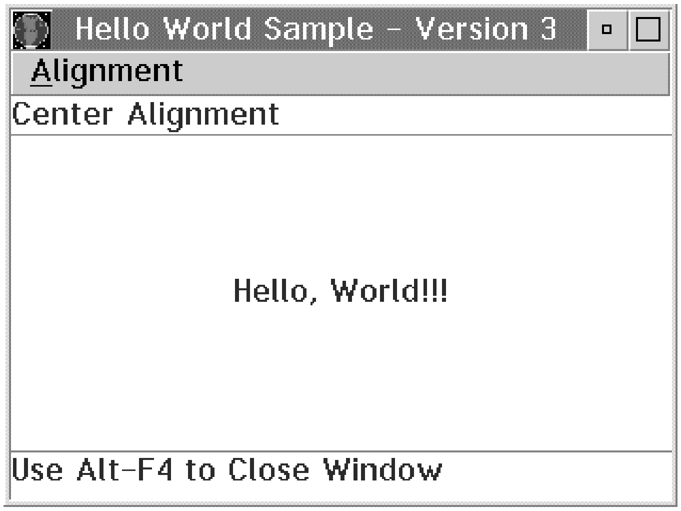
```
IApplication::current().userResourceLibrary()
```

You can explicitly specify the resource library on the IFrameWindow constructor by using the const IResourceId argument, giving an IResourceLibrary value.

If a required resource is not found, an exception is thrown, and the frame window is not constructed.

See "Adding Styles" on page 314 for more information on setting styles. For a list of the styles provided with IFrameWindow, refer to the *Open Class Library Reference*.

Figure 25 shows the components of a frame window created using the IFrameWindow class with the default style and some added controls.



Center Alignment

Hello, World!!!

Use Alt-F4 to Close Window

Figure 25. Frame Window Components

The following example defines a frame window:

```
//*************************************************************************
// framewin.cpp.                                                          *
//*************************************************************************
#include <iframe.hpp>
#include <iapp.hpp>
⋮
//*************************************************************************
// Main Routine - application entry point                                 *
//*************************************************************************
int main()
{

//*****************************************************
// Construct the frame window
//*****************************************************
IFrameWindow frame("Basic Frame Window",
                    WID_MAIN,
                    IFrameWindow::defaultStyle());
frame.setFocus().show();

//*****************************************************
// Add event handling
//*****************************************************
IApplication::current().run();

return 0;

}
```

Figure 26 shows the frame window created using the preceding example.

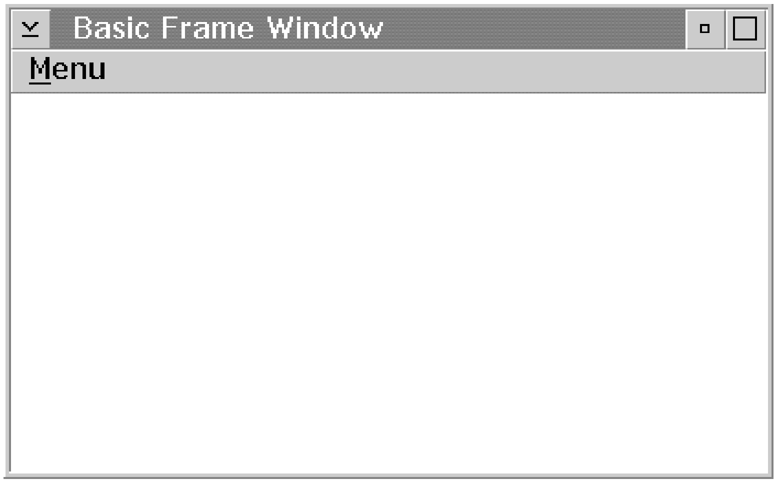


Figure 26. A Basic Frame Window

Frame Windows

When a frame window is minimized, the frame window hides and draws its minimized icon. Sometimes other windows associated with the frame window are drawn on top of its icon. This occurs when the windows are children of the frame window but not the client window or frame extensions.

To suppress the drawing of child windows, that are not frame extensions or the client window, when they are supposed to be minimized, add a handler to the frame window that detects when the frame is minimized, and hides these windows. The handler should make these windows visible when the frame is restored.

The User Interface Class Library defers positioning and sizing components of a frame window until the frame window shows. As a result, if you query the size and position of the frame window's client window or frame extensions, an accurate value is not returned until the frame window is shown.

The IWindow::show or IFrameWindow::showModally member functions automatically update the frame window. You can force the frame window to update itself by calling the IFrameWindow::update member function.

Changing the Title Bar

The *title bar* is the area at the top of each frame window that contains a window title.

You can specify the icon which displays when the application is minimized using the minimizedIcon style when you create the frame window.

If you do not provide a window title, the User Interface Class Library sets the title to a string loaded from the application's resource library. The ID of the string in the string table is the frame window's ID. If the User Interface Class Library cannot find a string, the title defaults to the system-generated title (typically, the name of the executable file).

The following code, from Hello World version 3, shows you how to specify a minimized icon and the window title when you create the frame window.

1. The title text is defined in the resource file:

   ```
   ⋮
   ICON WND_MAIN ahellow3.ico                    //Application icon

   STRINGTABLE
     BEGIN
       STR_HELLO,  "Hello, World!!!"             //Hello World string
       WND_MAIN,   "Hello World Sample - Version 3"   //Main window title string
   ⋮
   ```

   WND_MAIN is the frame window identifier. The frame window uses the window identifier (windowId) passed on the constructor to load its icon, title, menu bar, or accelerator table resources if these components are specified in the frame window style.

2. The following code comes from the AHELLOW3.CPP file:

```
⋮
int main()
{
  AHelloWindow mainWindow (WND_MAIN);
  mainWindow.setAlignment(AHelloWindow::left);
  mainWindow.sizeTo(ISize(400,300));
  mainWindow.setFocus();
  mainWindow.show();
  IApplication::current().run();
  return 0;
} /* end main */
⋮
AHelloWindow :: AHelloWindow(unsigned long windowId)
  : IFrameWindow(IFrameWindow::defaultStyle() |
                 IFrameWindow::minimizedIcon,
                 windowId)
⋮
```

When the application creates the AHelloWindow object, it constructs the IFrameWindow base class using the default style with a minimized icon, AHELLO3.ICO, and "Hello World Sample - Version 3" as the title text. The frame window determines the icon and title text based on the window ID and the resource library.

Adding a Menu Bar

The *menu bar* is the area near the top of a window, below the title bar and above the client area of the window. A menu bar contains a list of choices. When a user selects a choice on a menu bar, a pull-down menu associated with that choice is displayed.

The following sample contains a menu bar with only one submenu named **Alignment**. When you run the sample and select **Alignment**, the pull-down menu is displayed. The choices in the pull-down menu are **Left**, **Center**, and **Right**. When you select one of the choices, the text string in the client window aligns to the selected position and a check mark appears beside the selected item.

1. The following code, from the AHELLOW3.RC file, defines the text for the menu bar and its associated pull-down menu.

```
⋮
MENU WND_MAIN                                  //Main window menu bar
  BEGIN
        SUBMENU " Alignment", MI_ALIGNMENT      //Alignment submenu
          BEGIN
            MENUITEM " Left",       MI_LEFT     //Left menu item - F7 Key
            MENUITEM " Center",     MI_CENTER   //Center menu item - F8 Key
            MENUITEM " Right",      MI_RIGHT    //Right menu item - F9 Key
        END
  END
⋮
```

2. Use a command handler to implement the functions to be performed as a result of a user selecting a menu item. To place a check mark beside the appropriate menu item, create an IMenuBar object to represent the application menu bar.

   The following code, from the AHELLOW3.HPP file, shows the AHelloWindow class containing an ACommandHandler object, which is derived from the ICommandHandler class:

   ```
   ⋮
     private:
       IMenuBar      menuBar;
       IStaticText   statusLine;
       IStaticText   hello;
       IInfoArea     infoArea;
       ACommandHandler commandHandler;
   ⋮
   };
   #endif
   ```

3. The code in this section is from the AHELLOW3.CPP file.

   The following code creates the mainWindow object from the AHelloWindow class.

   ```
   ⋮
     AHelloWindow mainWindow (WND_MAIN);
   ⋮
   ```

   This code shows the first part of the AHelloWindow constructor. The menu bar defined in the resource file is associated with this AHelloWindow object.

   ```
   ⋮
   AHelloWindow :: AHelloWindow(unsigned long windowId)
     : IFrameWindow(IFrameWindow::defaultStyle() |
                    IFrameWindow::minimizedIcon,
                    windowId)
      ,menuBar(WND_MAIN, this)
      ,statusLine(WND_STATUS, this, this)
      ,hello(WND_HELLO, this, this)
      ,infoArea(this)
      ,commandHandler(this)
   {
   ⋮
   ```

   The following code attaches the command handler to the frame window.

   ```
   ⋮
     commandHandler.handleEventsFor(this);
   ⋮
   ```

   When the user selects a menu item, the ACommandHandler::command function is called to select the appropriate frame window function to call.

```
⋮
  switch (cmdEvent.commandId()) {
    case MI_CENTER:
      frame->setAlignment(AHelloWindow::center);
      break;
⋮
```

This code from the AHelloWindow::setAlignment function places the check mark beside the appropriate menu item.

```
⋮
  case center:
    hello.setAlignment(
      IStaticText::centerCenter);
    statusLine.setText(STR_CENTER);
    menuBar.checkItem(MI_CENTER);
    menuBar.uncheckItem(MI_LEFT);
    menuBar.uncheckItem(MI_RIGHT);
    break;
```

Creating an Information Area

The *information area* is a small rectangular area that is usually located at the bottom of a frame window. You can use the information area to display:

- A brief explanation of the state of an object
- Information about the completion of a process
- Information messages displayed with fly over help

Use the IInfoArea class to create and manage the information area. Objects of IInfoArea class provide a frame extension to show information about the menu item where the cursor is positioned. The string displayed in the information area is defined in a string table in the resource file.

The following sample uses the IInfoArea class to create the information area and the text to display in it.

1. The menu bar and string table are defined in the AHELLOW3.RC file. The string table contains strings of text, and each string is associated with a menu item. When you choose the menu item, the string related to that item displays in the information area.

```
⋮
MENU WND_MAIN                                //Main window menu bar
  BEGIN
        SUBMENU "~Alignment", MI_ALIGNMENT      //Alignment submenu
          BEGIN
            MENUITEM "~Left",       MI_LEFT     //Left menu item - F7 Key
            MENUITEM "~Center",     MI_CENTER   //Center menu item - F8 Key
            MENUITEM "~Right",      MI_RIGHT    //Right menu item - F9 Key
          END
  END
⋮
STRINGTABLE
```

```
  BEGIN
⋮
    STR_INFO,   "Use Alt-F4 to Close Window"    //Information area string
    MI_ALIGNMENT,"Alignment Menu"               //InfoArea - Alignment menu
    MI_CENTER,  "Set Center Alignment"          //InfoArea - Center menu item
    MI_LEFT,    "Set Left Alignment"            //InfoArea - Left menu item
    MI_RIGHT,   "Set Right Alignment"           //InfoArea - Right menu item
⋮
  END
```

2. This code is from the AHELLOW3.HPP file. The highlighted lines add an information area object to the AHelloWindow class.

```
⋮
class AHelloWindow : public IFrameWindow
{
public:
    AHelloWindow(unsigned long windowId);
    ~AHelloWindow();
⋮
  private:
    IMenuBar      menuBar;
    IStaticText   statusLine;
    IStaticText   hello;
    IInfoArea     infoArea;
    ACommandHandler commandHandler;
};
```

3. In the AHELLOW3.CPP file, construct the information area when AHelloWindow is created.

```
⋮
AHelloWindow :: AHelloWindow(unsigned long windowId)
  : IFrameWindow(IFrameWindow::defaultStyle() |
                 IFrameWindow::minimizedIcon,
                 windowId)
   ,menuBar(WND_MAIN, this)
   ,statusLine(WND_STATUS, this, this)
   ,hello(WND_HELLO, this, this)
   ,infoArea(this)
   ,commandHandler(this)
{
⋮
  hello.setText(STR_HELLO);
  infoArea.setInactiveText(STR_INFO);
⋮
```

Adding Styles

A *style* affects the appearance and behavior of a window. Each window class has styles that are encapsulated in style objects.

Generic styles are defined in IWindow and IControl. Classes derived from IWindow and IControl can combine their own styles with those of IWindow and IControl.

Styles

Each window class maintains its own default style object. You can access default style objects using the static member function defaultStyle and then set it using the static member function setDefaultStyle. Each window class also maintains a style object called classDefaultStyle that corresponds to the initial setting of defaultStyle.

Most window classes provide one or more constructors that accept a style object as one parameter. You can only construct a style object from existing style objects. These style objects are only used by window constructors. The style of a window can subsequently be changed and queried using the window class member functions. Also, some styles cannot change after a window has been created, in which case, no member function is provided to change the style.

The following sections describe how you can use bitwise operators. For the sake of simplicity, the IComboBox class and its styles are used for all examples. The IBitFlag class provides bitwise operators that you can use with the styles of the User Interface Class Library just as if you were using them with numbers. See IComboBox in the *Open Class Library Reference* for more information about this class and its styles.

Copying Styles

The assignment operator (=) returns one style object that is set equal to the specified style object. The value of the resulting object is equal to the value of the operand object. For example:

```
IComboBox::Style
  myStyle = IComboBox::dropDownType;
```

Combining Styles

The bitwise OR (|) operator returns a style object that is a combination of two style objects. The value of the resulting object is the bitwise OR of the value of the two operand objects.

You can combine any existing style objects, such as myStyle1 and myStyle2 in the following example, to create yet another style object. For example:

```
IComboBox::Style
  myStyle3 = myStyle1 | myStyle2;
```

This example adds the tabStop style to myStyle object:

```
IComboBox::Style myStyle = IComboBox::dropDownType;
myStyle |= IControl::tabStop;
```

In many cases, you can combine styles of one class with those of another class. Here, an IComboBox style is combined with an IControl style. The documentation for each class that has styles specifies whether other classes have compatible styles that you can use when constructing objects for those classes.

Testing Styles

The bitwise AND (&) operator returns an unsigned long integer that identifies if there are any bits common to the operand style or attribute objects. Typically, you use this operator to test whether a bitwise OR (|) operator has been used to combine one style object with another. For example:

```
Boolean isADropDown = false;
if (myStyle1 & IComboBox::dropDownType)
   isADropDown = true;
```

Negating Styles

The bitwise NOT (˜) operator returns a negated style object. The value of the resulting object is the bitwise NOT of the value of the operand object. For example:

```
IComboBox::Style::NegatedStyle
  negatedStyle = ˜myStyle;
```

This code returns an object named negatedStyle that negates the value of the myStyle object.

The precedence of the AND operator (&) is greater than the OR operator (|). You must be aware of operator precedence to avoid creating invalid styles that might not be obvious.

Note: If you do not want to consider operator precedence, specify the styles you want instead of negating others from the default User Interface Class Library styles.

The following example creates an invalid style that the IViewPort constructor will reject. This causes the following:

```
IViewPort::defaultStyle()
  | IViewPort::alwaysHorizontalScrollBar
  & ˜IViewPort::asNeededHorizontalScrollBar
```

to be evaluated as:

```
IViewPort::defaultStyle()
  | (IViewPort::alwaysHorizontalScrollBar
  & ˜IViewPort::asNeededHorizontalScrollBar)
```

as opposed to the following:

```
(IViewPort::defaultStyle()
  | IViewPort::alwaysHorizontalScrollBar)
  & ˜IViewPort::asNeededHorizontalScrollBar
```

Therefore, you must consider the order and the operator precedence when you negate a style because the User Interface Class Library cannot change the order in which operators are evaluated in the code statement.

Setting Window Styles

You can create a window with a specific style in the following ways:

- Create a window using a constructor that accepts the style as a parameter. The following three examples illustrate this method.

  This example shows how to create an entry field control with a style that is a combination of styles from IWindow, IControl, and IEntryField.

  ```
  IEntryField  entryField( ID_EF1, parent, owner,
                           IRectangle(10, 10, 100, 20),
                           IWindow::visible    |
                           IControl::tabStop   |
                           IControl::group     |
                           IEntryField::margin |
                           IEntryField::autoScroll );
  ```

  Alternatively, you can explicitly construct the style object and pass it as a parameter:

  ```
  IEntryField::Style efStyle = IWindow::visible    |
                               IControl::tabStop   |
                               IControl::group     |
                               IEntryField::margin |
                               IEntryField::autoScroll ;
  IEntryField  entryField( ID_EF1, parent, owner,
                           IRectangle(10, 10, 100, 20),
                           efStyle );
  ```

  You can also access the default style object using the static member function defaultStyle. This simplifies the preceding example to:

  ```
  IEntryField  entryField( ID_EF1, parent, owner,
                           IRectangle(10, 10, 100, 20),
                           IEntryField::defaultStyle()  |
                           IControl::tabStop            |
                           IControl::group );
  ```

- Use the static member function setDefaultStyle to set the default style and then construct the window. For example:

  ```
  IEntryField::Style efStyle = IEntryField::defaultStyle()  |
                               IControl::tabStop            |
                               IControl::group     ;
  IEntryField::setDefaultStyle(efStyle);
  IEntryField  entryField( ID_EF1, parent, owner,
                           IRectangle(10, 10, 100, 20) );
  ```

- Create a window with the default style and change it using member functions of the window. The example now becomes:

  ```
  IEntryField  entryField( ID_EF1, parent, owner,
                           IRectangle(10, 10, 100, 20) );
  entryField.enableGroup();              // Member function of IControl
  entryField.enableTabStop();            // Member function of IControl
  entryField.enableAutoScroll();         // Member function of IEntryField
  ```

For a complete list of available styles, see the *Open Class Library Reference*.

Using Cursor Classes

Window classes that can contain one or more items generally provide a nested cursor class. The *cursor classes* provide member functions to move through the items, either forward or backward.

Note: The User Interface Class Library cursor classes are designed to have the same look and feel as the Collection Class library cursors.

A cursor must be in a valid state to access the items in a list. A cursor is generally created in an invalid state. Any cursor function that causes the cursor to point to an item in the list validates the cursor. For example, the function setToFirst causes the cursor to be valid if there are items in the list. If the contents of the list that the cursor is iterating through change by the addition or removal of items, the cursor becomes invalid. It cannot be used to access items in the list until it is validated again by a function that points the cursor at a valid item.

Note: IWindow::ChildCursor is an exception to this.

Some cursors support iteration over items in a collection that match a particular property. For example, the constructor for a list box cursor can have a second parameter that determines whether the cursor returns all items in the list box or just the selected items.

The following example, from the Hello World version 5 AHELLOW5.CPP file, shows how to set text from the first selected item in a multiple-selection list box:

```
⋮
AHelloWindow &
  AHelloWindow :: setTextFromListBox()
{
  /*---------------------- Set Hello Text from ListBox ---------------------|
  | Create a cursor to the list box.  Using the default filter for a       |
  |   list box cursor, selectedItems, causes the setToFirst() function     |
  |   to position the cursor to the first selected item.                   |
  | Set the hello IStaticText control text value.                          |
  |-----------------------------------------------------------------------*/
  IListBox::Cursor lbCursor(listBox);
  lbCursor.setToFirst();
  hello.setText(listBox.elementAt(lbCursor));

  return (*this);                      //Return a reference to the frame
}
⋮
```

Specifying Message Box Information

A *message box* is a frame window that an application uses to display a note, caution, or warning to the user. For instance, an application can use a message box to inform a user of a problem that the application encountered while performing a task. The

Message Box

User Interface Class Library provides an IMessageBox class for displaying messages in a message box.

Creating a Message Box

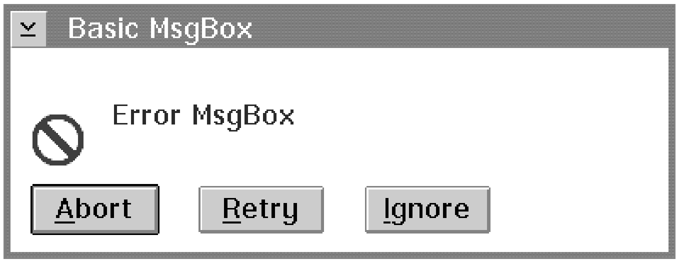
You construct objects of the IMessageBox class by using an object of a class derived from IWindow. The IWindow object becomes the owner of the new message box, as follows:

```
IMessageBox  messageBox(owner);
```

The following example shows you how to create a message box:

```
⋮
/**********************************************************/
/* Create the information type message box               */
/**********************************************************/
int about()
{
  IMessageBox msg(IWindow::desktopWindow());
  msg.setTitle("Basic MsgBox");
  msg.show("Could not open file:",
         IMessageBox::abortRetryIgnoreButton |
         IMessageBox::defButton1 |
         IMessageBox::errorIcon |
         IMessageBox::moveable);
  return 0;
}
```

Figure 27 shows a message box similar to the one created by the preceding example.



Error MsgBox

Figure 27. Example of a Message Box

Message Box

29 Creating and Using Text Controls

A *control* is a part of the user interface that lets a user interact with data.

This chapter explains how to code the following text controls:

- Static text
- Entry fields
- Multiple-line edit (MLE) fields
- Buttons
  - Push buttons
  - Radio buttons
  - Check boxes
  - Three-state check boxes

Controls are usually identified by text; for example, headings, labels in push buttons, field prompts, and titles in windows.

Creating a Static Text Control

Static text controls are text fields, bit maps, icons, and boxes that you can use to label or box other controls. Your user typically does not interact with these controls using the keyboard or mouse. Generally, you do not need to change a static control's appearance on the screen, so visually they tend to be unchanging. Static text control classes include IBitmapControl, IIconControl, IOutlineBox, IGroupBox, and IStaticText. The IStaticText class creates and manages the static text control window.

You can set the text and its color, size, and position in the static text window.

Refer to the *Open Class Library Reference* for a list of the public members provided with IStaticText as well as informaton on the other static control classes.

The following sample comes from the AHELLOW1.CPP file from the Hello World sample application and shows how to create a static text control.

```
⋮
IStaticText hello(IC_FRAME_CLIENT_ID, &mainWindow, &mainWindow);
hello.setText("Hello, World!!!");
hello.setAlignment(IStaticText::centerCenter);
mainWindow.setClient(&hello);
⋮
```

The first line uses the window ID, the parent window, and the owner window to create the static text control and an object for it.

The second line sets a text string in the control using the setText member function, which is inherited from ITextControl.

The third line uses the setAlignment member function to center the static text.

Figure 28 shows the Hello World version 1 static text control.

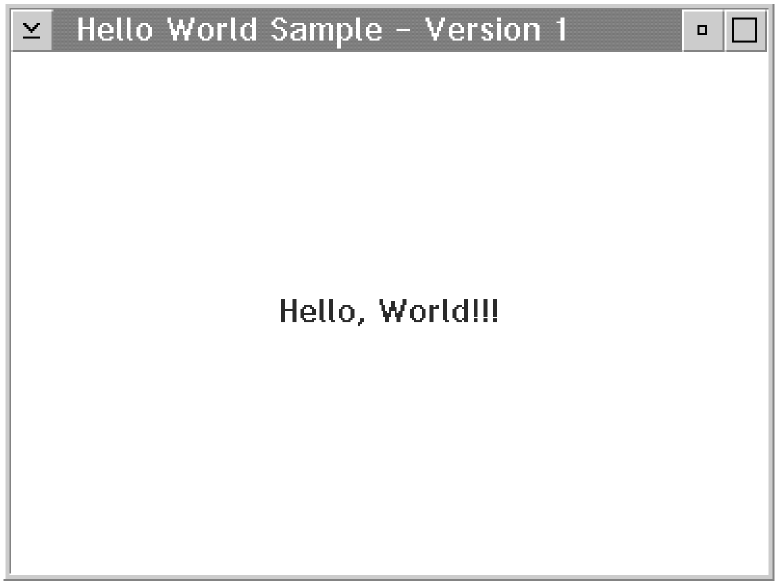


Figure 28. Hello World Version 1 Static Text Control

Understanding Entry Fields

An *entry field* is a control window that enables a user to view and edit a single line of text. An entry field provides the text-editing capabilities of a simple text editor and is useful whenever an application requires a short line of text from the user.

If the application requires more sophisticated text-editing capabilities and multiple lines of text from the user, the application can use a Multiple-line edit field. Refer to "Viewing and Editing Multiple-Line Edit (MLE) Fields" on page 326 for more information about MLE controls.

Applications typically use entry fields in dialog windows, although they can be used in non-dialog windows as well. The following section contains a sample and an example to show you how to create an entry field.

Creating an Entry Field

The following is an example of creating an entry field.

1. Declare three entry fields in the .hpp file, as follows:

```
#include <ientryfd.hpp>
/**********************************************************/
/*  Create the frame window                               */
/*   Declare the entry fields                             */
/**********************************************************/
class AppWindow : public IFrameWindow
{
public:
      AppWindow(unsigned long windowId);
      ~AppWindow();
      void handleEvents(unsigned long eventtype);
private:
      ICanvas         canvas;
      IEntryField     ef1;
      IEntryField     ef2;
      IEntryField     ef3;
      ACommandHandler * commandHandler;
};
```

2. Construct the entry fields in the .cpp file and manipulate data inside them.

```
⋮
/************************************************************/
/* Window constructor                                       */
/************************************************************/

#include "entryfd.h"
#include "entryfd.hpp"

//*****************************************************
//  Define a customized style to use in the third entryfd
//*****************************************************
IEntryField::Style efStyle = IWindow::visible  |
                             IControl::tabStop |
                             IControl::group   |
                             IEntryField::margin |
                             IEntryField::autoScroll;

AppWindow :: AppWindow(unsigned long windowId)
           : IFrameWindow("Entry Field Example",
                          windowId,
                          IFrameWindow::defaultStyle() |
                          IFrameWindow::menuBar),
             canvas(ID_CANVAS, this, this),
             ef1(ID_ENTRY, &canvas,&canvas,
                     IRectangle( 10, 200, 600, 240)),
             ef2(ID_ENTRY2, &canvas, &canvas,
                     IRectangle( 10, 125, 300, 175)),
             ef3(ID_ENTRY3, &canvas, &canvas,
                     IRectangle( 10, 50, 300, 100),
                      efStyle)
{
```

```
⋮

//*******************************************************
//  Create first entry field
//*******************************************************
  ef1.setLimit(50);
  ef1.setText("Initial Text for Entry Field 1");
  ef1.setBackgroundColor(IColor::yellow);
  ef1.setForegroundColor(IColor::red);
  ef1.setBorderColor(IColor::green);
  ef1.setFocus();

//*******************************************************
//  Create second entry field
//*******************************************************
  ef2.setLimit(50);
  ef2.setText("Initial Text for Entry Field 2");
  ef2.setBackgroundColor(IColor::yellow);
  ef2.setForegroundColor(IColor::red);
  ef2.setBorderColor(IColor::green);

//*******************************************************
//  Create third entry field
//*******************************************************

  ef3.setLimit(50);
  ef3.setText("Initial Text for Entry Field 3");
  ef3.setBackgroundColor(IColor::yellow);
  ef3.setForegroundColor(IColor::red);
  ef3.setBorderColor(IColor::green);

  moveSizeTo(IRectangle(0, 0, 670, 350));
  canvas.setBackgroundColor(IColor::blue);
  setClient(&canvas);
  show();
  commandHandler = new ACommandHandler(this, &ef1, &ef2, &ef3);
  commandHandler->handleEventsFor(this);
}
```

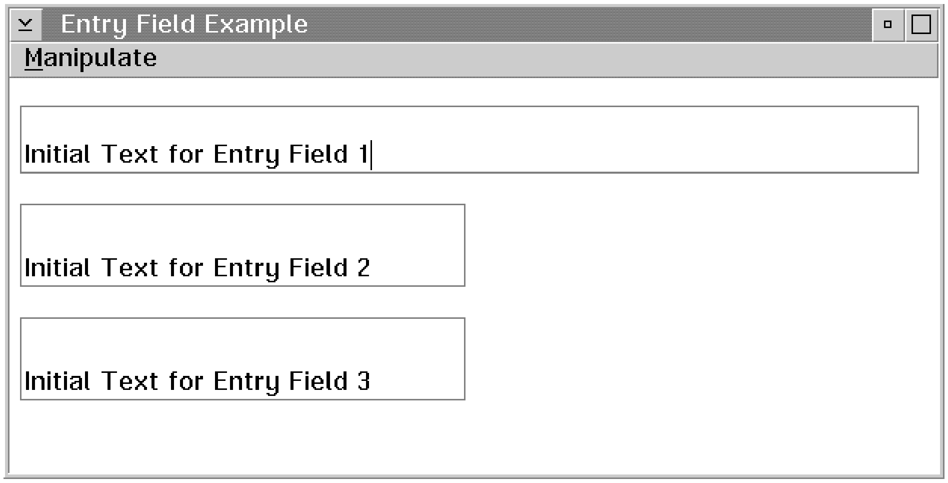
3. Declare a command handler in the .hpp file, as follows:

```
class ACommandHandler : public ICommandHandler {

public:
  ACommandHandler(AppWindow *efWindow,
                  IEntryField *ef1,
                  IEntryField *ef2,
                  IEntryField *ef3);

protected:
  virtual Boolean command(ICommandEvent& cmdEvent);

private:
  AppWindow *ef;
  IEntryField *ef1,
              *ef2,
              *ef3;
};
```

4. Add event handling to manipulate the data inside the entry fields:

```
ACommandHandler::ACommandHandler(AppWindow *efWindow,
                                 IEntryField *ef1,
                                 IEntryField *ef2,
                                 IEntryField *ef3):
    ef(efWindow),
    ef1(ef1),
    ef2(ef2),
    ef3(ef3)
{
}


IBase::Boolean ACommandHandler::command(ICommandEvent& cmdEvent)
{
  switch (cmdEvent.commandId())
   {
     case ID_READONLY_ITEM:
      ef1->enableDataUpdate(ef1->isWriteable());
      break;
     case ID_COPY_ITEM:
      if (ef1->hasSelectedText()) {
        ef1->copy();
      }
        break;
     case ID_CUT_ITEM:
      if (ef2->hasSelectedText()) {
        ef2->cut();
       }
        break;
     case ID_PASTE_ITEM:
      if (ef3->clipboardHasTextFormat()) {
        ef3->paste();
        }
        break;
     case ID_CLEAR_ITEM:
      if (ef3->hasSelectedText()) {
        ef3->clear();
        }
      break;
      return true;
   } // end of switch
    return false;
}
```

Figure 29 shows the entry fields created using the preceding example.



Initial Text for Entry Field 1

Initial Text for Entry Field 2

Initial Text for Entry Field 3

Figure 29. Entry Field Example

Viewing and Editing Multiple-Line Edit (MLE) Fields

A *multiple-line edit* (MLE) field enables users to view and edit multiple lines of text. Use the IMultiLineEdit class to create an MLE field. The member functions of the IMultiLineEdit class enable you to display text files with horizontal and vertical scrolling, read a file into and save it from an MLE, or perform basic clipboard tasks (for example, cut, paste, copy, and clear).

Creating an MLE

To create an object of the IMultiLineEdit class, include the ID of a specified MLE, the parent and owner windows, an IRectangle object, and one or more styles.

Styles define such functions as scrolling text, wrapping words, adding a border, and making the field read-only.

Refer to *Open Class Library Reference* for further information about the IMultiLineEdit class and its styles.

To create an MLE, use the following steps:

1. Declare an application frame window that contains an MLE and a command handler that can process events for the MLE.

```
/********************************************************/
/* Create the command handler                           */
//*******************************************************/
class ACommandHandler : public ICommandHandler {

public:
  ACommandHandler(AppWindow *mleWindow, IMultiLineEdit *amle);

protected:
  virtual Boolean command(ICommandEvent& cmdEvent);

private:
  IMultiLineEdit *mle;
};


/************************************************************/
/*  Create the frame window                                 */
/************************************************************/
class AppWindow : public IFrameWindow
 {
  public:
    AppWindow( unsigned long windowId);
    ~AppWindow();

  private:
    about();
    ITitle          title;
    IMultiLineEdit   mle;
    ACommandHandler  * commandHandler;
 };
```

2. The following code constructs the MLE in the .cpp file:

```
⋮
/**************************************************************/
/*                Window Constructor                          */
/**************************************************************/

AppWindow :: AppWindow( unsigned long windowId)
   : IFrameWindow(windowId,          // create Frame window
                  defaultStyle() | menuBar),
     title(this,"MLE Example"),      // include Title
     mle(ID_MLE, this, this)         // create MLE
{
   setClient(&mle);
   handleEventsFor(this);
   setIcon(ID_ICON);
   mle.setFocus();

//**********************************************************
// Create Command Handler
//**********************************************************
```

```
   commandHandler = new ACommandHandler(this, &mle);
   commandHandler->handleEventsFor(this);
```

⋮

3. Use the following code for event handling:

```
/**********************************************************/
/*         Command Handler Constructor                    */
/**********************************************************/
ACommandHandler :: ACommandHandler (AppWindow *mleWindow,
                                    IMultiLineEdit *amle):
  mle(amle)
{
}

/**********************************************************/
/* MyWindow Command Event Handler                         */
/**********************************************************/
IBase::Boolean ACommandHandler :: command(ICommandEvent& cmdevt)
 {
   switch (cmdevt.commandId())
   {
      case ID_IMPORT_ITEM:
         mle->importFromFile("import.txt",IMultiLineEdit:: MLEFormat);

         return true;

      case ID_EXPORT_ITEM:
         mle->exportToFile("export.Txt",IMultiLineEdit:: noTran);
         return true;

      case ID_INIT_ITEM:
         mle->setText("This is some initial text.");
         return true;

      case ID_MARK_ITEM:
         mle->selectRange(IRange(13,19));
         return true;

      case ID_COPY_ITEM:
         if (mle->hasSelectedText())
           mle->copy();
         return true;

      case ID_CUT_ITEM:
         if (mle->hasSelectedText())
           mle->cut();
         return true;

      case ID_PASTE_ITEM:
         if (mle->clipboardHasTextFormat())
           mle->paste();
         return true;

      case ID_DELMARK_ITEM:
         if (mle->hasSelectedText())
           mle->discard();
         return true;
      case ID_DELALL_ITEM:
         mle->removeAll();
         return true;

      case ID_INSERT_ITEM:
         mle->add("inserted");
         return true;

      case ID_WORDWRAP_ITEM:
         mle->enableWordWrap(!mle->isWordWrap());
         return true;

      case ID_HOME_ITEM:
         mle->setCursorPosition(0);
         return true;

   } /* end switch */
   return false;
}
```

Loading and Saving a File

The following member functions from the IMultiLineEdit class allow you to import text to an MLE from a file and export text from an MLE into a file.

Member Function	Use To
importFromFile	Load a file into an MLE
exportToFile	Save from an MLE
exportSelectedTextToFile	Save marked text in an MLE into a file

Refer to the *Open Class Library Reference* for descriptions of these member functions.

You can load and save a file to the MLE, as follows:

```
⋮
    case ID_IMPORT_ITEM:
      mle->importFromFile("import.txt",IMultiLineEdit::MLEFormat);
      return true;

    case ID_EXPORT_ITEM:
      mle->exportToFile("export.txt",IMultiLineEdit::noTran);
      return true;
⋮
```

Positioning the Cursor

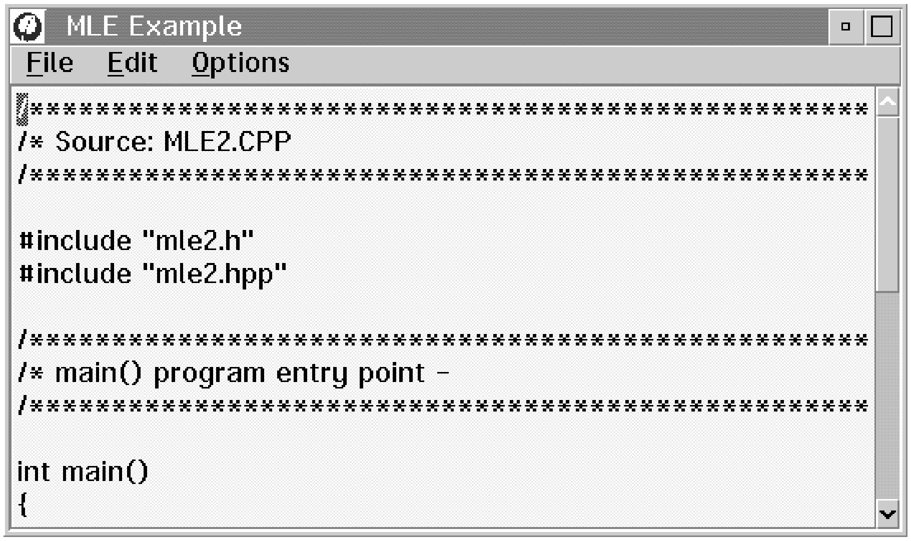
You can position the cursor on a specific line of an MLE or in a specific character position, add to or remove lines from an MLE, or request the number of lines in an MLE.

Refer to the *Open Class Library Reference* for descriptions of MLE member functions.

Position the cursor on the first line, as follows:

```
case ID_HOME_ITEM:
   mle->setCursorPosition(0);
   return true;
```

Figure 30 shows the cursor on the first line of the MLE.



```
/*************************************************
/* Source: MLE2.CPP
/*************************************************

#include "mle2.h"
#include "mle2.hpp"

/*************************************************
/* main() program entry point -
/*************************************************

int main()
{
```

Figure 30. Example of Positioning the Cursor on the First Line

Performing Clipboard Operations

The IMultiLineEdit class has several member functions to perform clipboard operations, including copy, cut, paste, clear, and discard. After you define an MLE, use these member functions to copy text to the clipboard, cut and put text into the clipboard, or paste only the marked lines from the clipboard. Refer to the *Open Class Library Reference* for descriptions of other member functions.

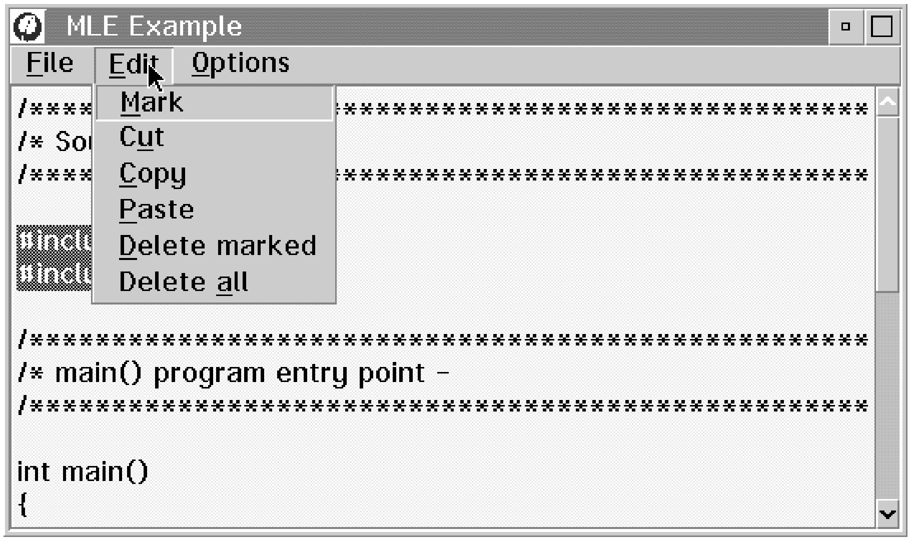
The following code performs clipboard operations:

```
⋮
  case ID_MARK_ITEM:            // First mark some text
    mle->selectRange(IRange(13,19));
    return true;

  case ID_COPY_ITEM:
    if (mle->hasSelectedText())
```

```
      mle->copy();
      return true;

  case ID_PASTE_ITEM:
    if (mle->clipboardHasTextFormat())
       mle->paste();
    return true;
```

⋮

Figure 31 shows an example of cutting text to the clipboard. It contains marked lines in the client area and a menu option, **Edit**, with six menu items, including **Cut**. The menu item ID of the **Cut** menu item is ID_CUT_ITEM.



```
/****                ***********************************
/* So
/****                ***********************************

#inclu
#inclu

/*****************************************************
/* main() program entry point -
/*****************************************************

int main()
{
```

Figure 31. Example of Cutting Text to the Clipboard

The following statements show you how to implement the ID_CUT_ITEM member function:

⋮

```
  case ID_CUT_ITEM:             // Check that text is marked
    if (mle->hasSelectedText())       // then cut it to the clipboard
       mle->cut();
    return true;
```

⋮

Creating Buttons

A *button* is a type of control window used to initiate an operation or to set the attributes of an operation. A button can appear alone or with a group of other buttons. When buttons are grouped, you can move from button to button within the group by pressing the **Arrow** keys. You can also move among groups by pressing the **Tab** key.

A user can select a button by clicking it with the mouse or by pressing the **spacebar** when the button has the keyboard focus. In most cases, a button changes its appearance when selected.

A button is always owned by another window, usually a dialog window or an application's client window. It posts messages or sends notification messages to its owner when a user selects the button. The owner window receives messages from a button and can send messages to the button to alter its position, appearance, and enabled or disabled state.

To use a button in a dialog window, your application specifies the control in a dialog template in the application's resource-definition file. The application processes button messages in the dialog-window procedure.

Understanding Button Types

There are four main types of buttons, which determine how the button looks and behaves:

- Push buttons
- Radio buttons
- Check boxes
- Three-state check boxes

A radio button, check box, or three-state check box *control* an operation; a push button *initiates* an operation. For example, you might set printing options (such as paper size, print quality, and printer type) in a print-command dialog window containing an array of radio buttons and check boxes. After setting the options, you select a push button to notify an application that printing should begin (or be canceled). Then the application queries the state of each check box and radio button to determine the printing parameters.

The following sections discuss the different types of buttons in more detail.

Creating a Push Button

A *push button* is a rectangular window that contains a text string. Typically, an application uses a push button to let the user start or stop an operation. A push button represents an action that is initiated when a user selects it. You can label it with text, graphics, or both. When a user selects a push button, the action occurs immediately if there is a handler for the generated command event.

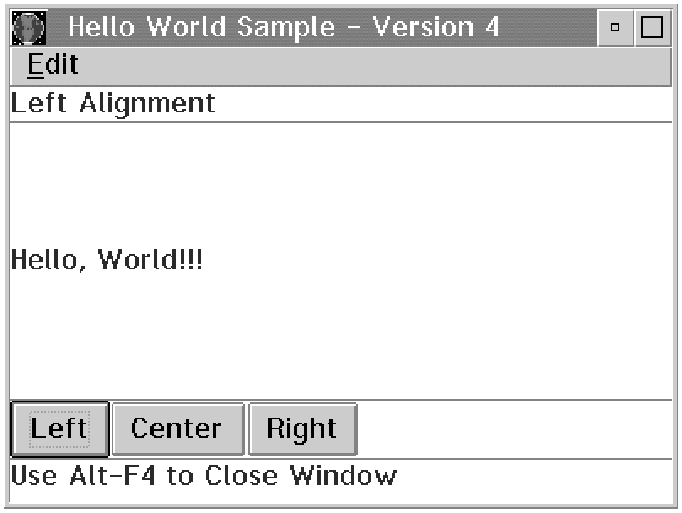
Use the IPushButton class to create and maintain the push button window. By default, a push button generates an application ICommandEvent. You can change the default style by changing the window style value to generate a help event or a system command event. Using system command events is not recommended for portable applications.

Refer to *Open Class Library Reference* for a list of the styles provided for IPushButton and for the IPushButton derived class, IGraphicPushButton.

The Hello World version 4 application defines three push buttons (**Left**, **Center**, and **Right**) in the AHELLOW4.CPP file.

The command events generated by pressing the **Left**, **Center**, and **Right** push buttons are handled by the AHelloWindow::command member function. Note, the command events are the same as those used for the corresponding menu items. Therefore, the command function processing is the same whether you press the push button or select the item on the menu bar.

Figure 32 shows the Hello World version 4 push buttons.



Hello World Sample - Version 4

Left Alignment

Hello, World!!!

Use Alt-F4 to Close Window

Figure 32. Hello World Version 4 Push Buttons

Creating a Radio Button

A *radio button* is a window with text displayed to the right of a small circular indicator. Use radio buttons to display a set of choices from which the user can select one. Each time the user selects a radio button, that button's state toggles between *selected* and *unselected*. This state remains until the next time the user selects the button. An application typically uses radio buttons in groups.

A group of radio buttons contains at least two radio buttons. Within a group, usually one button is selected by default. The user can move the selection to another button by using the cursor keys; however, only one button can be selected at a time. Radio buttons are appropriate if an *exclusive* choice is required from a fixed list of options. For example, applications often use radio buttons to let users select the screen foreground and background colors.

The IRadioButton class lets you create and manage the radio button window. The ISelectHandler class processes the selection of a radio button. You add the handler to either the radio button or its owner window by calling the handler's handleEventsFor function.

Refer to Chapter 38, "Adding Events and Event Handlers" on page 467 for information about event handlers.

The following example creates a group of radio buttons.

The text associated with each radio button is defined in the resource file as string text, as follows:

```
⋮
STRINGTABLE
  BEGIN
    STR_BLACK,  "Black"
    STR_WHITE,  "White"
    STR_BLUE,   "Blue"
    STR_RED,    "Red"
    STR_YELLOW, "Yellow"
  END
⋮
```

1. Declare the radio buttons in the .hpp file.

```
/***********************************************************/
/* Command handler class declaration                       */
/***********************************************************/
class MyCommandHandler : public ICommandHandler {

public:
  MyCommandHandler(AppWindow *mainWindow);

protected:
  virtual Boolean command(ICommandEvent& cmdEvent);

private:
   AppWindow *appWindow;

};
```

```
/*********************************************************/
/* Select handler class declaration                      */
/*********************************************************/
class MySelectHandler: public ISelectHandler
 {
  public:
        MySelectHandler(IStaticText *info);
  protected:
        selected(IControlEvent& evt);
  private:
   Boolean fProcess;
   IStaticText *staticText;
  };

/*********************************************************/
/* AppWindow declaration                                 */
/*********************************************************/
class AppWindow : public IFrameWindow
 {
  public:
    AppWindow( unsigned long windowId);
    ~AppWindow();
    AppWindow & enableButton();
    AppWindow & disableButton();
   private:

    ITitle           * title;
    ICanvas          * canvas1;
    IGroupBox        * groupBox;
    IStaticText      * staticText;
    IRadioButton     * white;
    IRadioButton     * black;
    IRadioButton     * blue;
    IRadioButton     * red;
    IRadioButton     * yellow;
    MySelectHandler   * selectHandler;
    MyCommandHandler  * commandHandler;

   };
```

2. Declare the buttons in the .cpp file.

```
AppWindow :: AppWindow( unsigned long windowId)
   : IFrameWindow(windowId,
                 defaultStyle() |
                 menuBar)
    {

  //*******************************************************
  // Create Title
  //*******************************************************
  title = new ITitle(this,ID_RADIO_TITLE);

  //*******************************************************
  // Create Canvas
  //*******************************************************
  canvas1 = new ICanvas(ID_CANVAS, this, this);
  moveSizeTo(IRectangle(10,10,650,540));
```

```
setClient(canvas1);

IWindow * pParent= canvas1;
IWindow * pOwner = canvas1;


//*********************************************************
// Create Status Area
//*********************************************************
staticText = new IStaticText(ID_TEXT,
                             canvas1,
                             canvas1,
                             IRectangle(20,425,470,450));
staticText->setText(ID_STAT_TITLE);

//*********************************************************
// Create Group Box
//*********************************************************
groupBox = new IGroupBox(ID_GROUPBOX,
                        canvas1,
                        canvas1,
                        IRectangle( 90,10,610,400));
groupBox->setText("Color Selection");

//*********************************************************
// Create Radio Buttons
//*********************************************************
white = new IRadioButton(WND_WHITEBT, pParent, pOwner, IRectangle(100,300,250,360));
white->setText(STR_WHITE);

black = new IRadioButton(WND_BLACKBT, pParent, pOwner, IRectangle(100,230,250,290));
black->setText(STR_BLACK);

blue = new IRadioButton(WND_BLUEBT, pParent, pOwner, IRectangle(100,160,250,220));
blue->setText(STR_BLUE);

red = new IRadioButton(WND_REDBT, pParent, pOwner, IRectangle(100,90,250,150));
red->setText(STR_RED);

yellow = new IRadioButton(WND_YELLOWBT, pParent, pOwner, IRectangle(100,20,250,80));
yellow->setText(STR_YELLOW);


//*********************************************************
// Set the group style of the controls
//*********************************************************
white->enableGroup().enableTabStop();

//*********************************************************
// Select white as the default button
//*********************************************************
white->select();

//*********************************************************
// Set the select handler to handle events
//*********************************************************
selectHandler = new MySelectHandler(staticText);
selectHandler->handleEventsFor(white);
```

```
  selectHandler->handleEventsFor(black);
  selectHandler->handleEventsFor(blue);
  selectHandler->handleEventsFor(red);
  selectHandler->handleEventsFor(yellow);

  //**********************************************************
  // Set the command handler to handle menu events
  //**********************************************************
  commandHandler = new MyCommandHandler(this);
  commandHandler->handleEventsFor(this);

  setFocus().show();
}
```

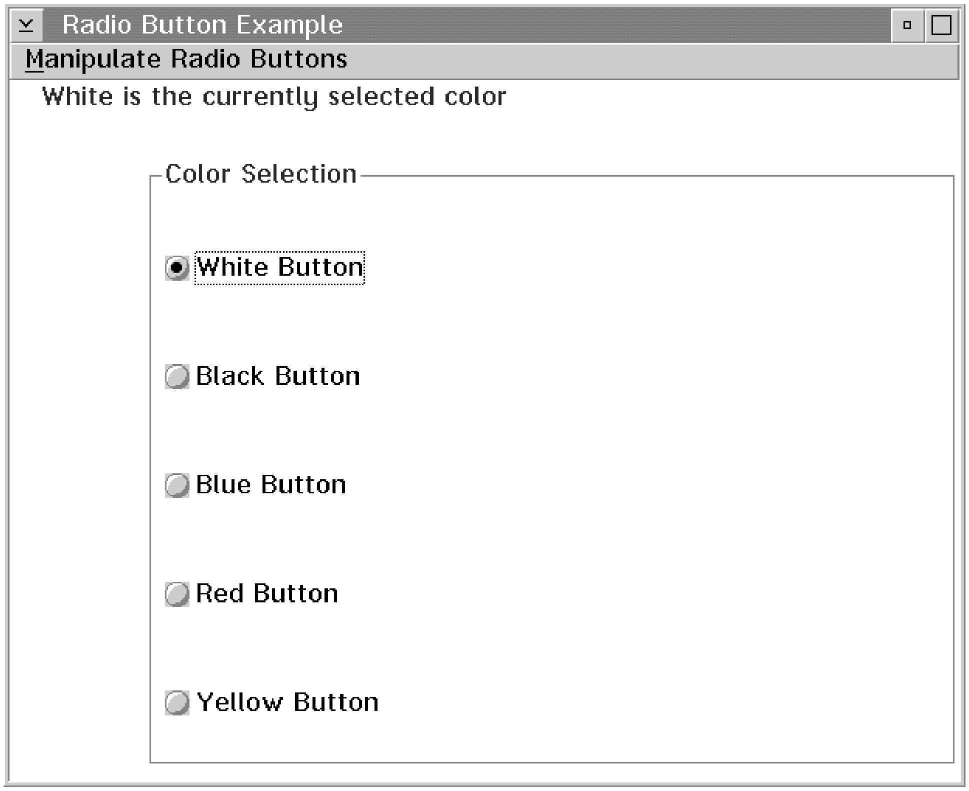
3. Process menu events in the command handler routines.

```
⋮
/**************************************************************/
/* Construct the command handler                              */
/**************************************************************/
MyCommandHandler::MyCommandHandler(AppWindow *mainWindow)
{
  appWindow = mainWindow;
}

/**************************************************************/
/* MyWindow command event handler                             */
/**************************************************************/
IBase::Boolean MyCommandHandler :: command(ICommandEvent& cmdevt)
 {
   switch (cmdevt.commandId())
   {
      case ID_DISABLE_BLUE_BTN:
        appWindow->disableButton();
          break;
      case ID_ENABLE_BLUE_BTN:
        appWindow->enableButton();
          break;
   }
   return false;
}

/**************************************************************/
/* Enable the radio button                                    */
/**************************************************************/
AppWindow & AppWindow::enableButton()
{
  blue->enable();
  return (*this);
}

/**************************************************************/
/* Disable the radio button                                   */
/**************************************************************/
AppWindow & AppWindow::disableButton()
{

   blue->disable();
   return (*this);
}
⋮
```

4. Process selection events in the select handler routines.

```
⋮
/*********************************************************/
/* MyWindow Select Event Handler                         */
/*********************************************************/
MySelectHandler::MySelectHandler(IStaticText *info)
   :ISelectHandler(),
    staticText(info)
{
}

/*********************************************************/
/* Set static text when radio button selected.           */
/*********************************************************/
IBase::Boolean MySelectHandler::selected(IControlEvent& evt)
{

  Boolean fprocess = false;
  switch(evt.controlId())
  {
  case WND_BLACKBT:
       staticText->setText("Black is the currently selected color");
       fProcess=false;
       break;

  case WND_WHITEBT:
       staticText->setText("White is the currently selected color");
       fProcess=false;
       break;

  case WND_REDBT:
       staticText->setText("Red is the currently selected color");
       fProcess=false;
       break;

  case WND_BLUEBT:
       staticText->setText("Blue is the currently selected color");
       fProcess=false;
       break;

  case WND_YELLOWBT:
       staticText->setText("Yellow is the currently selected color");
       fProcess=false;
       break;
  }
  return fProcess = false;
}
```

Figure 33 shows the radio button created with the preceding code example.



White is the currently selected color

Color Selection

● White Button

○ Black Button

○ Blue Button

○ Red Button

○ Yellow Button

Figure 33. Radio Buttons

Creating a Check Box

Check boxes are similar to radio buttons, except that they can offer *multiple-choice* selection, as well as individual choice. When a user selects the choice, a check mark symbol (√) appears in the check box to indicate that the choice is selected. By selecting the choice again, the user deselects the check box. Use a check box to set a choice in a group of choices that are not mutually exclusive.

Check boxes also toggle application features *on* or *off*. For example, a word processing application might use a check box to let the user turn word wrapping on or off.

The ICheckBox class lets you create and maintain a check box. The selection of a check box is processed by using the ISelectHandler class. You add the handler to either the check box or its owner window.

Refer to Chapter 38, “Adding Events and Event Handlers” on page 467 for information about event handlers.

The following example shows you how to create a check box:

1. Create a check box by first making a declaration in the .hpp file, as follows:

```
⋮
/***********************************************************/
/* Set canvas class declaration                            */
/***********************************************************/
class MySet : public ISetCanvas
{
  public:
    MySet(unsigned long winId, IWindow* pParent);

  private:
    ICheckBox   check1;
    ICheckBox   check2;
    ICheckBox   check3;
};
⋮

/***********************************************************/
/* Application Window declaration                          */
/***********************************************************/
class AppWindow : public IFrameWindow
 {
  public:
    AppWindow( unsigned long windowId);
    ~AppWindow();

  private:
    ITitle            title;
    MySet           * pSetCv;
    IMultiCellCanvas   canvas;
 };
```

2. Construct the frame window.

```
⋮
/*************************************************************/
/* Window Constructor                                        */
/*************************************************************/

AppWindow :: AppWindow( unsigned long windowId)
   : IFrameWindow(windowId,
                  defaultStyle()),
     title(this,PSZ_OBJECT,PSZ_VIEW),
     canvas(ID_CANVAS, this, this)
{

setClient(&canvas);

//*********************************************************
// Create Canvas
//*********************************************************
pSetCv = new MySet(ID_SET, &canvas);
canvas.addToCell(pSetCv, 2, 2);

}
⋮

⋮
/**********************************************************/
/* MySet constructor                                      */
/**********************************************************/
MySet :: MySet(unsigned long winId, IWindow* pParent)
 : ISetCanvas(winId, pParent, pParent),
   check1(ID_BOX1, this, this, IRectangle(),
                    ICheckBox::classDefaultStyle |
                    IControl::group),
   check2(ID_BOX2, this, this),
   check3(ID_BOX3, this, this)
{
   check1.setText(PSZ_BOX1);
   check2.setText(PSZ_BOX2);
   check3.setText(PSZ_BOX3);
   setText(PSZ_GROUP);
}
⋮
```

Figure 34 shows the check box created using the preceding example.

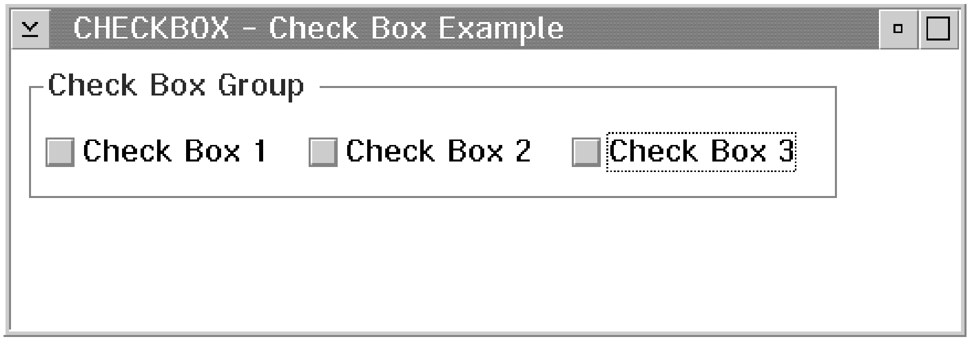


Figure 34. Check Box Example

Creating a Three-State Check Box

Three-state check boxes are similar to check boxes, except that they can be displayed in *halftone* as well as selected and unselected. An application might use the halftone state to indicate that, currently, the check box is not selectable. The selection of a three-state check box is processed by using the ISelectHandler class and adding the handler to either the three-state check box or its owner window.

If a three-state check box is selected, it can be either checked or halftoned. You have to use a combination of the isSelected and isHalftone member functions to determine the state of a three-state check box. The following table shows the state of the three-state check box and its relation to the isSelected and isHalftone member functions.

Current State	isSelected	isHalftone
checked	true	false
halftone	true	true
not checked	false	false

The following example shows how to create a three-state check box:

```
I3StateCheckBox three(ID_THREE, &canvas, &canvas,
                      IRectangle(100,220,250,280));
three.setText("3 State");
⋮
three.selectHalftone();
⋮
if (three.isSelected())  {      //Determine state of the button
   if (three.isHalftone())  {
      // is halftone
   } else {
      // is checked
   }  /* end if */
} else {
   // is not selected
}  /* end if */
```

30

Creating and Using List Controls

You can use lists in your User Interface Class Library application by including the following controls:

- List boxes
- Combination boxes
- Sliders
- Spin buttons

Understanding List Box Controls

A *list box* is a control that displays several items at a time, one or more of which can be selected by the user.

An application uses a list box when it requires a list of selectable fields that is too large for the display area or a list of choices that can change dynamically. Each list item contains a text string and an optional handle. The text string is displayed in the list box window, but the handle is available to the application to reference other data associated with each of the items in the list.

The IBaseListBox class creates and manages list box control windows. Two types of list box controls are derived from IBaseListBox. They are IListBox and ICollectionViewListBox. IListBox extends the IBaseListBox list box control creation and management to include adding, removing, and replacing list box items. ICollectionViewListBox<Element, Collection> template class extends the IBaseListBox control to enabling viewing of an ordered collection as items in a list box. The sequence of elements is the same between the ordered collection and the list box.

You can attach an ISelectHandler to a list box or its owner window to process events created when the user selects or double-clicks on an item in the list box. Typically, the owner is a dialog window or the client window of an application frame window.

Using List Boxes

You can use a list box to display a list in a window. Notification messages are sent from the list box to its owner window, enabling the application to respond to user actions in the list. Events are routed first to the list box, then to its owner.

Once you create the list box, your application controls the inserting and deleting of list items. Items can be inserted at the end of the list, automatically sorted into the list, or inserted at a specified index or cursor position. You can add an array of items at a specified index or cursor position.

For an ICollectionViewListBox list box, the inserting, sorting, and deleting actions occur on the collection.

You can use cursors to manipulate the list box. Cursors can be filters to process all items in the list box or only the selected ones.

Creating a List Box

This section shows you how to create an IListBox list box control. The sample comes from Hello World version 5. It does the following:

- Creates a list box
- Uses the addAscending member function
- Uses a select handler
- Creates a list box cursor with the default filter, a selected items filter

This code comes from the AHELLOW5.CPP file:

```
⋮
 ,listBox(WND_LISTBOX, &clientWindow, &clientWindow, IRectangle(),
             IListBox::defaultStyle() |
             IControl::tabStop |
             IListBox::noAdjustPosition)
⋮
/************************************************/
/* Add items to the list box                    */
/************************************************/

for (int i=0;i<HI_COUNT;i++ )
   listBox.addAscending(HI_WORLD+i);
selectHandler.handleEventsFor(&listBox);
⋮
/************************************************/
/* Create a cursor                              */
/************************************************/
IListBox::Cursor lbCursor(listBox);
lbCursor.setToFirst();
/*************************************************/
/* Set the text to the first item in the list box*/
/*************************************************/
hello.setText(listBox.elementAt(lbCursor));
⋮
```

Adding or Deleting a List Box Item

Your applications can add or delete an item in a list box. Items in a list are specified with a 0-based index (beginning at the top of the list). A new list is created empty; then, the application initializes the list by inserting items.

The application specifies the text and position for each new item. It can specify an absolute-position index or use a list box cursor.

For an ICollectionViewListBox control, the list box control window reflects actions on the associated collection. So an element removed from the collection, will be visually reflected in the list box control.

The following example shows you how to create an IListBox list box control and then add and delete items. This declaration shows a frame window that has a list box.

1. Declare a frame window with the list box as a child in the .hpp file:

   ```
   ⋮
   class Frame : public IFrameWindow
   {
    public:
       Frame(unsigned long windowId);
       ~Frame();
       void handleEvent(unsigned long int);

    private:
       ITitle            title;
       ICanvas           canvas;
       IEntryField       ef;
       IListBox          listbox;
       IStaticText       stTxt1;
       IStaticText       stTxt2;
       ACommandHandler   * commandHandler;
   };
   ⋮
   ```

2. Construct the frame window, initializing the child controls. The frame is made owner of the canvas and title, and all other controls are children of the canvas.

   ```
   ⋮
   /*********************************************************/
   /* Construct the frame window                            */
   /*********************************************************/
   Frame::Frame(unsigned long windowId)
      : IFrameWindow(windowId,
                     IFrameWindow::defaultStyle() |
                     IFrameWindow::menuBar),
        title(this,"List Box Example"),
        canvas(ID_CANVAS,this,this),
        ef(ID_ENTRY,          &canvas, &canvas, IRectangle(10, 355, 600, 400)),
        listbox(ID_LISTBOX,   &canvas, &canvas, IRectangle(10, 10, 600, 300)),
        stTxt1(ID_STTXT1,     &canvas, &canvas, IRectangle(10, 420, 600, 400)),
        stTxt2(ID_STTXT2,     &canvas, &canvas, IRectangle(10, 325, 600, 345))
    {
   ```

3. Handle commands from the menu bar using a customized command handler. In the constructor for the ACommandHandler class (which is a subclass of ICommandHandler), make a local copy of the Frame object so its handleEvent function can be called. Then, in the ACommandHandler::command function,

look for the specific items from the menu bar. When application-specific items are found, route them to the Frame::handleEvent function for processing, as follows:

```
ACommandHandler::ACommandHandler(Frame *listWindow)
{
  list = listWindow;
}


void Frame :: handleEvent(unsigned long int eventtype)
 {
  switch (eventtype)
  {
// Add item to listbox
    case ID_ADD_ITEM:
      if (!ef.isEmpty()) {
        listbox.addAsFirst(ef.text());
        ef.setText("");
      }
     break;
// Add item in ascending order to list box
    case ID_ASC_ITEM:
      if (!ef.isEmpty()) {
        listbox.addAscending(ef.text());
        ef.setText("");
      }
     break;

// Add item in descending order to list box
    case ID_DESC_ITEM:
      if (!ef.isEmpty()) {
        listbox.addDescending(ef.text());
        ef.setText("");
      }
     break;

// Delete selected item(s) from list box
    case ID_DEL_ITEM:
      {
        IListBox::Cursor lbc(listbox, IListBox::Cursor::selectedItems);
        for (lbc.setToFirst(); lbc.isValid(); lbc.setToFirst()) {
          listbox.removeAt(lbc);
        }
      }
     break;

// Delete all items from list box
    case ID_DELALL_ITEM:
      if (!listbox.isEmpty()) {
        listbox.removeAll();
      }
     break;

// Allow only single selection in list box
    case ID_SINGLE_ITEM:
      listbox.disableMultipleSelect();
      listbox.disableExtendedSelect();
     break;

// Allow multiple selection in list box
    case ID_MULTI_ITEM:
      listbox.disableExtendedSelect();
      listbox.enableMultipleSelect();
     break;

⋮
IBase::Boolean ACommandHandler::command(ICommandEvent& cmdEvent)
{
  switch (cmdEvent.commandId())
  {
    case ID_ADD_ITEM:
    case ID_ASC_ITEM:
    case ID_DESC_ITEM:
    case ID_DEL_ITEM:
    case ID_DELALL_ITEM:
    case ID_SINGLE_ITEM:
    case ID_MULTI_ITEM:
    case ID_EXTEND_ITEM:
     list->handleEvent(cmdEvent.commandId());
   return true;
 } // end of switch
 return false;
}
```

Figure 35 shows the list box created with the preceding code example.

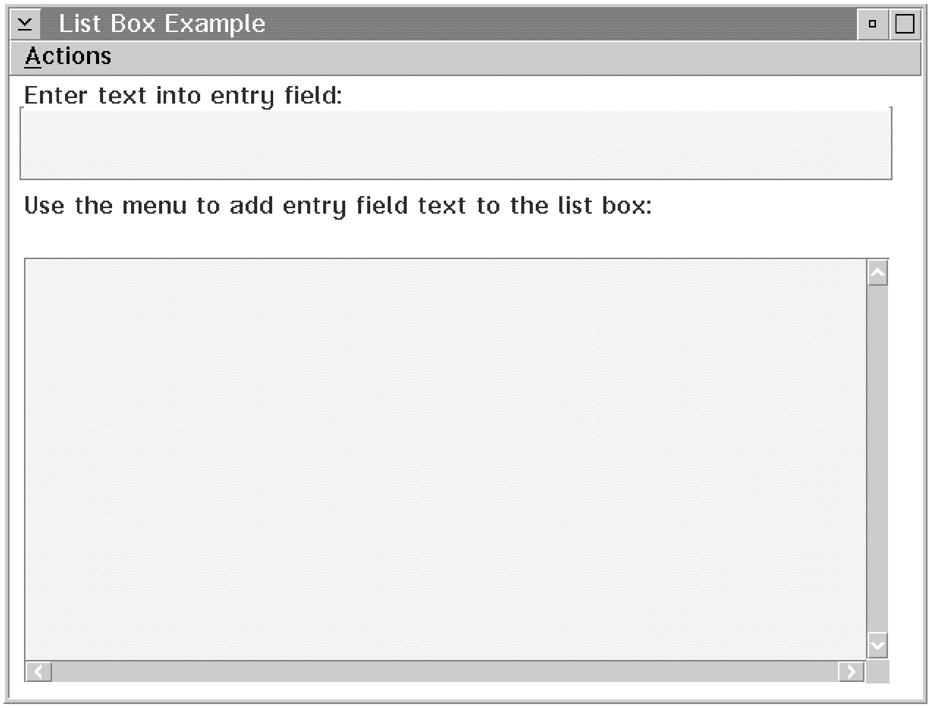

List Box Example
Actions
Enter text into entry field:
Use the menu to add entry field text to the list box:

Figure 35. A List Box

Understanding Combination Box Controls

A *combination box* is two controls in one: an entry field and a list box. There are three types of combination box controls:

- Simple
- Drop-Down
- Drop-Down List

This section describes how to create combination box controls, also called *combination boxes* and *prompted entry fields*, to let the user choose and edit items from a list in a PM application.

Combination box controls enable the user to enter data by typing in the entry field or by choosing an item in the list box, unless it is a drop-down list, in which case, you cannot edit the entry field. Combination box controls manipulate the list box using the following member functions:

- IComboBox::add

- IComboBox::remove
- IComboBox::replaceAt

A combination box control automatically manages the interaction between the entry field and the list box. For example, when the user chooses an item in the list box, the combination box control displays the text for that item in the entry field. Then, the user can edit the text without affecting the item in the list box. When the user types letters in the entry field, the combination box control scrolls the list box contents so that items with those letters become visible.

Objects of the IBaseComboBox class create and manage combination box control windows. There are two types of combination boxes derived from IBaseComboBox. They are IComboBox and ICollectionViewComboBox.

Population of the combination box list box items is done by the derived classes. IComboBox contains the add, remove, and replace functionality, while ICollectionViewComboBox populates the combination box list box from collection elements using the setItems member function.

Creating a Combination Box

This section shows you how to create a combination box control. The code comes from the Hello World version 6 sample application. The ADIALOG6.CPP file does the following:

- Creates a drop-down combination box using the following initializer in the ATextDialog constructor:

  ```
  ⋮
  ,textField( DID_ENTRY,&clientCanvas,&clientCanvas
     ,IRectangle(), IWindow::visible|IComboBox::dropDownType)
  ⋮
  ```

- Uses a loop to load strings from the resource file into the combination box in ascending order, as follows:

  ```
  ⋮
  for (int i=0;i<HI_COUNT;i++ )
     textField.addAscending(HI_WORLD+i);
  ⋮
  ```

- Loads the entry field portion of the combination box with the text string passed into the constructor, using the following code:

  ```
  ⋮
  textField.setText(saveText);
  ⋮
  ```

- Disables the entry field portion of the combination box for automatic scrolling, adds a margin, and sets a tab stop.

```
⋮
textField.disableAutoScroll().enableMargin().enableTabStop();
⋮
```

- Retrieves the text that users leave in the entry field portion of the combination box from the combination box object by using the text function, as follows:

```
⋮
saveText = textField.text();
⋮
```

Figure 36 shows the Hello World 6 combination box control.

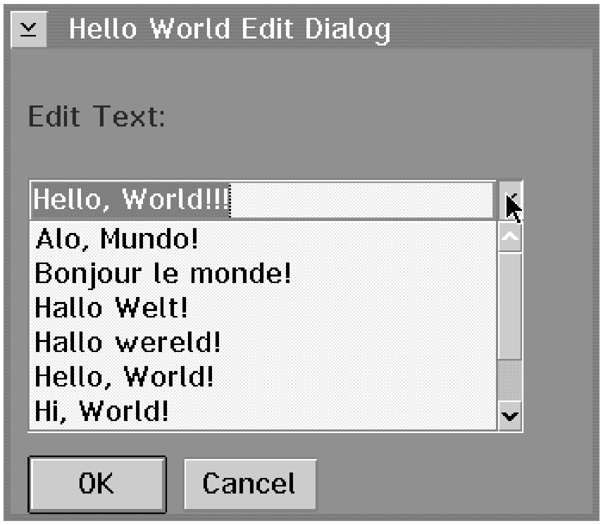


Figure 36. A Combination Box

Understanding Slider Controls

A *slider* is a visual component that enables a user to set, display, or modify a value by moving the slider arm along the slider shaft.

A slider consists of a slider arm, one or two slider scales and, optionally, detents, tick marks, tick text, and slider buttons. Note that you can have two slider scales, but only the primary one will be visible.

The following table lists the slider components and their descriptions:

Slider Component	Description
Detent	A user-selectable mark that can be placed anywhere along the slider scale.

Slider Component	Description
Progress indicator	A read-only version of a slider
Slider arm	An arm that shows the current value by its position on the slider shaft and can be changed programmatically, as well as by users. Users can move the arm along the shaft to set slider values.
Slider buttons	Buttons that move the slider arm incrementally in the indicated direction.
Slider shaft	A track for the slider arm to move along.
Tick text	A label indicating the value the tick mark represents.
Tick mark	An incremental value in a slider scale.

Typically, sliders let users set values that have familiar increments, such as feet, degrees, or decibels. You can use sliders for other purposes when immediate feedback is required, such as to blend colors or show a task's percentage of completion. For example, your application might let a user mix and match color shades by moving a slider arm, or a progress indicator (with the ribbonStrip style) could show how much of a task is complete by filling in the slider shaft as the task progresses.

The slider's appearance and user's interaction with a slider is similar to that of a scroll bar. However, these two controls are not interchangeable because each has a unique purpose. A scroll bar scrolls information into view that is outside a window's work area, while the slider sets, displays, or modifies that information.

You can customize a slider to meet varying application requirements, while providing a user interface component that can be used easily to develop applications that conform to the Common User Access (CUA) user interface guidelines. Your application can specify different scales, sizes, and orientations for its sliders, but the underlying function of the control remains the same.

The ISlider class inherits from the IProgressIndicator class, which is a read-only version of the slider control. Typically, you use a progress indicator to display the percentage of a task that is complete by filling in its shaft as the task progresses. The default for progress indicators is to use the ribbonStrip style to fill the shaft. If you do not use this, there is a slider arm present to indicate the current value. Users cannot move the slider arm in a progress indicator.

Attach a handler derived from IEditHandler to the slider to capture when a user moves the slider arm.

Creating a Slider Control

The following example is comprised of three sliders used to set the red, green, and blue colors in a color mixer. As the slider arm moves, the static text color will change appropriately. program.

1. Define the main window. A multi-cell canvas is used as the client window. The client canvas contains the sliders as well as a multi-cell canvas containing the static text which represents the current color.

```
class ColorMixerWindow : public IFrameWindow
  {
  public:
    ColorMixerWindow();
    ~ColorMixerWindow();
    ColorMixerWindow& displayNewColor();

 private:
    IMultiCellCanvas  canvas;
    ISlider           redSlider, greenSlider, blueSlider;
    IMultiCellCanvas  colorAreaCanvas;
    ISetCanvas        colorAreaFrame;
    IStaticText       colorArea;
    ColorMonitor      colorMonitor;
    ISetCanvas        redTitleCanvas, greenTitleCanvas, blueTitleCanvas;
    IStaticText       redTitle, greenTitle, blueTitle;
    IStaticText       redValue, greenValue, blueValue;
    IStaticText       mixerTitle;
  };
```

2. Define the ColorMonitor class. The ColorMonitor class is used to detect when a slider is moved. The edit member function is overridden to detect when a new color is to be displayed in the color area.

```
class ColorMonitor : public IEditHandler

  {
  public:
    ColorMonitor( ColorMixerWindow *colorMixerWindow )
      : _colorMixerWindow( colorMixerWindow )
      {;}

  protected:
    Boolean edit( IControlEvent &event );

  private:
    ColorMixerWindow *_colorMixerWindow;
  };
```

3. Create the main window.

```
ColorMixerWindow::ColorMixerWindow( )
  : IFrameWindow("Slider Example" )
  , canvas( ID_MCCANVAS, this, this )
  , mixerTitle( ID_MIXER_TITLE, &canvas, &canvas )
  , redTitleCanvas( ID_RED_CANVAS, &canvas, &canvas )
  , redValue( ID_RED_VALUE, &redTitleCanvas, &redTitleCanvas )
  , redTitle( ID_RED_TITLE, &redTitleCanvas, &redTitleCanvas )
  , redSlider(ID_RED_SLIDER, &canvas, &canvas, IRectangle(), 256 )
  , greenTitleCanvas( ID_GREEN_CANVAS, &canvas, &canvas )
  , greenValue( ID_GREEN_VALUE, &greenTitleCanvas, &greenTitleCanvas )
  , greenTitle( ID_GREEN_TITLE, &greenTitleCanvas, &greenTitleCanvas )
  , greenSlider(ID_GREEN_SLIDER, &canvas, &canvas, IRectangle(), 256 )
  , blueTitleCanvas( ID_BLUE_CANVAS, &canvas, &canvas )
  , blueValue( ID_BLUE_VALUE, &blueTitleCanvas, &blueTitleCanvas )
  , blueTitle( ID_BLUE_TITLE, &blueTitleCanvas, &blueTitleCanvas )
  , blueSlider(ID_BLUE_SLIDER, &canvas, &canvas, IRectangle(), 256 )
  , colorAreaCanvas( ID_COLOR_CANVAS, &canvas, &canvas )
  , colorAreaFrame( ID_COLOR_FRAME, &colorAreaCanvas, &colorAreaCanvas )
  , colorArea( ID_COLOR_AREA, &colorAreaCanvas, &colorAreaCanvas )
  , colorMonitor( this )

  {
  ⋮
```

4. Set up sliders and color area.

```
// Put a border of 10 around the sliders
canvas
  .setColumnWidth(1, 10 )
  .setColumnWidth(5, 10 )
  .setRowHeight( 1, 10 )
  .setRowHeight( 15, 10 )
// Mark the column that contains the sliders as expandable.
  .setColumnWidth(2, 10,true);

mixerTitle
  .setText( "Color Mixer" )
  .setAlignment( IStaticText::centerCenter );

// Set up the sliders to have a range of 0 to 255 for color selection
redSlider
  .setTickText(0, "0")
  .setTickText(255, "255")
  .moveArmToTick( 0 );
redTitle.setText("Red");
redValue
  .setText("0")
  .setLimit( 3 );
redTitleCanvas
  .setDeckOrientation( ISetCanvas::horizontal )
  .setPackType( ISetCanvas::tight );

greenSlider
  .setTickText(0, "0")
  .setTickText(255, "255")
  .moveArmToTick( 0 );
greenTitle.setText("Green");
greenValue
  .setText("0")
  .setLimit( 3 );
greenTitleCanvas
  .setDeckOrientation( ISetCanvas::horizontal )
  .setPackType( ISetCanvas::tight );

blueSlider
```

```
  .setTickText(0, "0")
  .setTickText(255, "255")
  .moveArmToTick( 0 )
  .setForegroundColor(IColor::white );
blueTitle.setText("Blue");
blueValue
  .setText("0")
  .setLimit( 3 );
blueTitleCanvas
  .setDeckOrientation( ISetCanvas::horizontal )
  .setPackType( ISetCanvas::tight );

// Set each slider's background to the color that the slider
// represents.
redSlider.setBackgroundColor   ( IColor::red );
greenSlider.setBackgroundColor( IColor::green );
blueSlider.setBackgroundColor  ( IColor::blue );

// Add a ISetCanvas to the multicell canvas and use its text feature to
// put a border around the colorArea. The IFont for the colorAreaFrame is
// used to figure out the size of the first row.
IFont colorAreaFont( &colorAreaFrame );
colorAreaCanvas
  .addToCell( &colorAreaFrame, 1, 1, 3, 3 )
  .addToCell( &colorArea, 2, 2 )
  .setColumnWidth( 2, 10, true )
  .setRowHeight( 1, 5 + colorAreaFont.maxCharHeight() )
  .setRowHeight( 2, 10, true );

colorAreaFrame.setText("Color Area");

// Set the slider visible ticks to be every 5th one.
for ( int i = 0; i <= 255; i=i+5 )
  {
  redSlider.setTickLength  ( i, 10 );
  greenSlider.setTickLength( i, 10 );
  blueSlider.setTickLength ( i, 10 );
  }

// Add the controls to the multicell canvas.
canvas
  .addToCell( &mixerTitle,      2, 2 )
  .addToCell( &redTitleCanvas,   2, 4 )
  .addToCell( &redSlider,        2, 6 )
  .setRowHeight( 6, 10, true )
  .addToCell( &greenTitleCanvas, 2, 8 )
  .addToCell( &greenSlider,      2, 10)
  .setRowHeight( 10, 10, true )
  .addToCell( &blueTitleCanvas,  2, 12)
  .addToCell( &blueSlider,       2, 14)
  .setRowHeight( 14, 10, true )
  .addToCell( &colorAreaCanvas,   4, 4, 1, 11);

// Add the colorMonitor to each slider so that we can detect when
// to update the colorArea.
colorMonitor
  .handleEventsFor( &redSlider )
  .handleEventsFor( &blueSlider )
```

```
    .handleEventsFor( &greenSlider );

  // Initialize the color areas color.
  colorArea.setBackgroundColor( IColor( 0, 0, 0 ));
  setClient(&canvas);
  }
```

5. Change color area to mix of colors specified by the sliders.

```
ColorMixerWindow& ColorMixerWindow::displayNewColor()

  {
  // Use the armTickOffset of each of the sliders to create an IColor
  // object to use to set the background color of the colorArea.
  IColor newColor( redSlider.armTickOffset(),
                   greenSlider.armTickOffset(),
                   blueSlider.armTickOffset() );
  colorArea.setBackgroundColor( newColor );

  // Display the value used to create the colorArea's background color.
  redValue.setText  ( IString( redSlider.armTickOffset())   );
  greenValue.setText( IString( greenSlider.armTickOffset()) );
  blueValue.setText ( IString( blueSlider.armTickOffset())  );

  return *this;
  }
```

6. Handle the events occuring when a slider's value changes.

```
IBase::Boolean ColorMonitor::edit( IControlEvent& event )

  {
  // When the slider's value changes display a new background color in the
  // colorArea.
  _colorMixerWindow->displayNewColor();
  return true;
  }
```

Figure 37 shows the slider created by the preceding example.

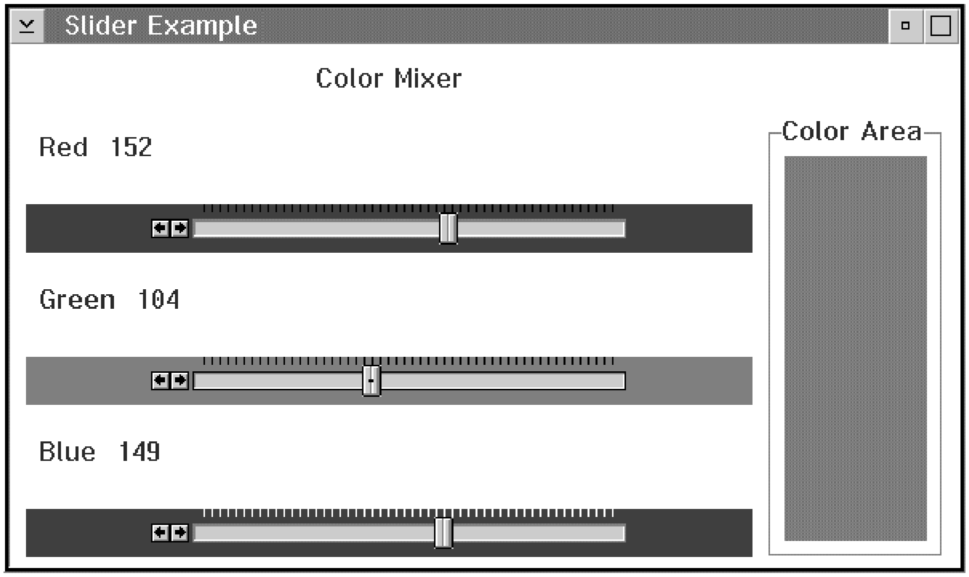


Figure 37. A Slider Example

Understanding Spin Buttons

A *spin button* control is a visual component that gives users quick access to a finite set of data by letting them select from a scrollable ring of choices. Because the user can see only one item at a time, a spin button should be used only with data that is intuitively related, such as the months of the year, or an alphabetic list of cities or states.

A spin button consists of at least one spin field and up and down arrows that are stacked on top of one another. These arrows are positioned to the right of the spin field.

Two types of spin buttons are derived from IBaseSpinButton. They are INumericSpinButton and ITextSpinButton.

You can create multi-field spin buttons for those applications in which users must select more than one value. For example, in setting a date, the spin button control can provide individual fields for setting the month, day, and year. The first spin field in the spin button could contain a list of months; the second, a list of numbers; and the third, a list of years.

The application uses a multi-field spin button by creating one master component that contains a spin field and the spin arrows, and servant components that contain only spin fields. The spin buttons are created during the construction of the master. When a servant spin field has the focus, it is spun by the arrows in the master component or by the cursor keys.

The value in a spin button entry field can be an element in an array of data or within a range of integers, defined by an upper and lower limit. Spin buttons which use arrays of data are text spin buttons and those that use a range of integers are numeric spin buttons.

Attach a handler derived from ISpinHandler to the spin button to capture spin events, such as the user pressing the up or down arrow.

Creating a Spin Button

The following example shows you how to create three spin buttons to show the three parts of a date (month, day, and year). It demonstrates how to initialize the data of both a text spin button and a numeric spin button. It also shows how to retrieve the value of the spin buttons and show the values in a message box. The spin buttons are children of the client canvas. The frame also contains a status area and a push button. The push button is added as an extension below the client canvas.

1. Define the main window in the .hpp file.

```
/*********************************************************/
/* Declare the frame window                              */
/*********************************************************/
class AppWindow : public IFrameWindow {
 public:
   AppWindow(unsigned long windowId);
   ~AppWindow();
   ITextSpinButton     * spinbtn1;
   INumericSpinButton  * spinbtn2,
                       * spinbtn3;
   IStaticText           statusarea;
   IPushButton           pushbtn;
   void eventHandle();

 private:
   ITitle          title;
   ICanvas         canvas;
   ACommandHandler * commandHandler
```

2. Create the spin buttons. Add the extensions and the command handler. in the .cpp file, as follows:

```
/*********************************************************/
/* Create the frame window                               */
/*********************************************************/
AppWindow::AppWindow(unsigned long windowId)
   : IFrameWindow(windowId,
     IFrameWindow::defaultStyle()),
     title(this, "Spin Button Example"),
     canvas(WID_CANVAS, this, this, IRectangle(5, 5, 410, 460)),
     statusarea(WID_STATUS, this, this),
     pushbtn(WID_BUTTON, this, this)

{
// Customize the push button with text
pushbtn.setText("OK");

//Create month spin button (text type spin button)
spinbtn1 = new ITextSpinButton(WID_MONTH,&canvas, &canvas,
                               IRectangle(10,250,200,350));

//Create day spin button (numeric type spin button)
spinbtn2 = new INumericSpinButton(WID_DAY,&canvas, &canvas,
                                  IRectangle(10,150,200,240 ));

//Create year spin button (numeric type spin button)
spinbtn3 = new INumericSpinButton(WID_YEAR,&canvas, &canvas,
                                  IRectangle(10,50,200,140 ));
⋮

//Add the status area as an extension
statusarea.setText("Select a MONTH, DAY and YEAR:");

setClient(&canvas);

addExtension(&statusarea, IFrameWindow::aboveClient,
     .05, IFrameWindow::thickLine);

//Add the push button as an extension
addExtension(&pushbtn, IFrameWindow::belowClient,
     .05, IFrameWindow::thickLine);

//Add the command handler
commandHandler = new ACommandHandler(this);
commandHandler->handleEventsFor(this);
}
```

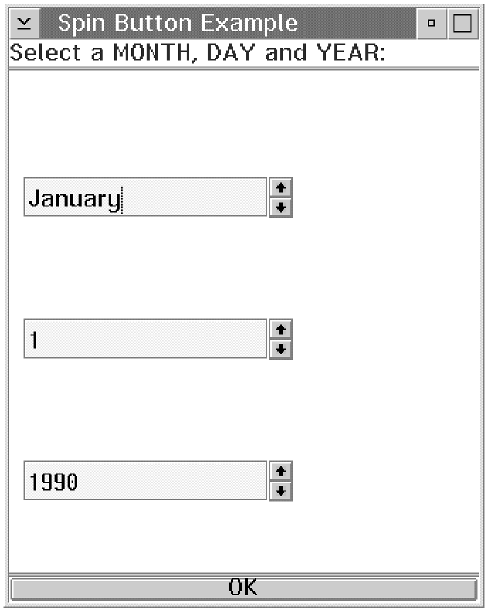
3. Add data to the spin buttons:

```
const int kArraySize = 12;
const int daySize = 31;
const int yearSize = 2000;

const char* textArray[kArraySize] = { "January", "February", "March",
       "April", "May","June", "July", "August", "September", "October",
       "November", "December" };

// Add month data to spin button
for ( int i = 0; i < kArraySize; i++ )
      spinbtn1->addAsLast( textArray[i]);

// Set range of days to day spin button
spinbtn2->setRange( IRange( 1, daySize ) );

// Set range of years to year spin button (1990 - 2000)
spinbtn3->setRange( IRange( 1990, yearSize ) );
```

4. Handle an event to show the data in the spin buttons, as follows:

```
// When OK push button is pressed - display the data inside the
// spin buttons
  switch (cmdEvent.commandId())
  {
    case WID_BUTTON:
      push->eventHandle();
      break;
⋮

void AppWindow :: eventHandle()
 {
  IString month = spinbtn1->text() += " ";
  IString day   = spinbtn2->value();
  IString year  = spinbtn3->value();
  IString text  = "You have selected: ";
  IString date  = text += month;
  date += day;
  date += " ";
  date += year;

// Display the data retrieved from the spin buttons
   IMessageBox msg(IWindow::desktopWindow());
   msg.setTitle("Spin Buttons Selection Notifier");
   msg.show(date, IMessageBox::informationIcon | IMessageBox::okButton)
```

Figure 38 shows the spin button created using the preceding example.



Select a MONTH, DAY and YEAR:

January

1

1990

Figure 38. Spin Button Example

Spin Buttons

31 Creating and Using Canvas Controls

A *canvas* is a window that manages its child windows. Different canvases provide a range of support, including the following:

- Managing the size and position of child windows
- Providing a movable split bars between windows
- Supporting the ability to scroll a window

With the canvas classes, you can build windows with multiple child controls that contain fixed-size areas, user-sizeable areas, and scrollable areas. In addition, a canvas control lets you control tabbing between child controls, providing an alternative to using dialog boxes.

Generally, you build a complex window with a canvas control as the client area. This canvas can contain other canvas controls to build the desired layout.

The canvas classes are:

- ICanvas
- ISplitCanvas
- ISetCanvas
- IToolBar
- IMultiCellCanvas
- IViewPort

The set and multiple-cell canvases automatically size and position their child windows for you, based on the child window's minimum size.

Understanding Split Canvases

A *split canvas* places its child controls into panes. The panes are separated by moveable or fixed split bars. (The default is movable split bars.) A split canvas can have its split bars oriented vertically or horizontally.

Note: In AIX, you can only move the split canvas using the small square buttons. However, in OS/2, you can move a split canvas using any part of the split bar line.

Use a split canvas to contain controls that can be resized to display more information, such as list boxes, containers, MLEs, and notebooks.

Note: Use the IListBox::noAdjustPosition style on a list box control in a split canvas, because otherwise the OS/2 operating system may adjust the height of the list box so it will not fill the height of the split canvas.

The order in which you create the child controls determines both their relative position on the split canvas and the order in which tab and cursor keys switch focus between them. For a canvas with vertical split bars, the child controls are arranged with the control that was created first in the leftmost pane. For a canvas with horizontal split bars, the control that was created first is placed in the top pane.

Creating a Split Canvas

The following example shows you how to create a split canvas.

1. Declare a split canvas in the .hpp file.

```
/**********************************************************/
/* MySplit class - split canvas class and associated      */
/* controls                                               */
/**********************************************************/
class MySplit : public ISplitCanvas
{
  public:
    MySplit(unsigned long winId, IWindow* pParent);

  private:
    IStaticText  statTxt1;
    IStaticText  statTxt2;
    IStaticText  statTxt3;
};

/**********************************************************/
/* Command handler declaration                            */
/**********************************************************/
class ACommandHandler : public ICommandHandler {

public:
  ACommandHandler(AppWindow *asplcan);

protected:
  virtual Boolean command(ICommandEvent& cmdEvent);

private:
  AppWindow *splcan;
};

/**********************************************************/
/* AppWindow class declaration                            */
/**********************************************************/
class AppWindow : public IFrameWindow
 {
  public:
    AppWindow( unsigned long windowId);
    ~AppWindow();
    AppWindow & updateCanvas(unsigned long eventtype);
  private:

    ITitle          *   title;
    MySplit         *   splitCv;
    ACommandHandler *   commandHandler;
 };
```

2. Create the window constructor in a .cpp file, as follows:

```
/***********************************************************/
/* Window constructor                                      */
/***********************************************************/

AppWindow :: AppWindow( unsigned long windowId)
   : IFrameWindow(windowId,                 // create Frame window
                  defaultStyle() | menuBar)
{

//*********************************************************/
// Create title                                            */
//*********************************************************/
title = new ITitle(this,PSZ_OBJECT,PSZ_VIEW);

//*********************************************************/
// Create split canvas                                     */
//*********************************************************/
splitCv = new MySplit(ID_SPLIT, this);
setClient(splitCv);

//*******************************************************
// Create command handler
//*******************************************************
commandHandler = new ACommandHandler(this);
commandHandler->handleEventsFor(this);
}

/***********************************************************/
/* MySplit constructor                                     */
/***********************************************************/
MySplit :: MySplit(unsigned long winId, IWindow* pParent)
 : ISplitCanvas(winId, pParent, pParent),
   statTxt1(ID_TEXT1, this, this),
   statTxt2(ID_TEXT2, this, this),
   statTxt3(ID_TEXT3, this, this)
{
   setSplitWindowPercentage(&statTxt1, 20);
   setSplitWindowPercentage(&statTxt2, 40);
   setSplitWindowPercentage(&statTxt3, 40);
   statTxt1.setText(PSZ_TEXT1);
   statTxt2.setText(PSZ_TEXT2);
   statTxt3.setText(PSZ_TEXT3);
   statTxt1.setBackgroundColor(IColor::red);
   statTxt2.setBackgroundColor(IColor::white);
   statTxt3.setBackgroundColor(IColor::blue);
   statTxt3.setForegroundColor(IColor::white);
}
```

3. Handle command events:

```
/**********************************************************/
/* Construct the command handler                          */
/**********************************************************/
ACommandHandler::ACommandHandler(AppWindow *asplcnv)
{
  splcan = asplcnv;
}

/**********************************************************/
/* MyWindow command event handler                         */
/**********************************************************/
IBase::Boolean ACommandHandler :: command(ICommandEvent& cmdEvent)
 {
   switch (cmdEvent.commandId())
   {
      case ID_VERT_ITEM:
      case ID_HORIZ_ITEM:
      case ID_DOUBLE_EDGE_ITEM:
      case ID_HALVE_EDGE_ITEM:
      case ID_DOUBLE_MIDDLE_ITEM:
      case ID_HALVE_MIDDLE_ITEM:
       splcan->updateCanvas(cmdEvent.commandId());
       return true;
   }
   return false;
}
```

Figure 39 shows a split canvas created by using the preceding example.

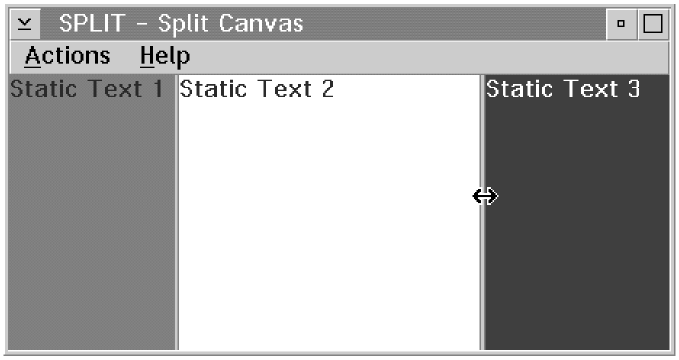


Figure 39. Split Canvas Example

The following examples show how to create a window containing two split canvases. Each pane is occupied by a static text control.

1. This code from the header file declares the ASplitCanvas class as a subclass of IFrameWindow.

Note: Member functions are initialized in the order that they appear in the class declaration.

```
#include <iframe.hpp>                          // IFrameWindow
#include <istattxt.hpp>                        // IStaticText
#include <isplitcv.hpp>                        // ISplitCanvas
class ASplitCanvas : public IFrameWindow
{
  public:
    ASplitCanvas(unsigned long windowId);        // Constructor

  private:
    ISplitCanvas  horzCanvas,             // The canvases will be created
                  vertCanvas;             // in the same order they
    IStaticText   lText,                        // are declared.
                  rText,
                  bText;
};
```

2. This code creates the window.

```
 1 ASplitCanvas :: ASplitCanvas( unsigned long windowId )
 2   : IFrameWindow( windowId )
 3   , horzCanvas( WND_CANVAS, this, this )
 4   , vertCanvas( WND_CANVAS2, &horzCanvas, &horzCanvas )
 5   , lText( WND_TXTL, &vertCanvas, &vertCanvas )
 6   , rText( WND_TXTR, &vertCanvas, &vertCanvas )
 7   , bText( WND_TXTB, &horzCanvas, &horzCanvas )
 8 {
 9
10   horzCanvas.setOrientation( ISplitCanvas::horizontalSplit ); //Give the canvas
11   setClient( &horzCanvas );          //a horizontal split bar
12                                      //and make it the client area
13
14   vertCanvas.setOrientation( ISplitCanvas::verticalSplit );//Give the canvas
15                                      //a vertical split bar
16   lText.setText(STR_TOPLEFT);        //Set top left static text
17   lText.setAlignment( IStaticText::centerCenter );
18
19   rText.setText(STR_TOPRIGHT);       //Set top right static text
20   rText.setAlignment( IStaticText::centerCenter );
21
22   bText.setText(STR_BOTTOM);         //Set bottom static text
23   bText.setAlignment( IStaticText::centerCenter );
24
25   setFocus().show();                 //Set focus and show window
26
27 } /* end ASplitCanvas :: ASplitCanvas(...) */
```

Lines 1 through 7 create a canvas with horizontal and vertical split bars. The canvases are created in the same order they were declared in the header file. In line 4, the vertical canvas is defined as a child of the horizontal canvas.

Lines 10 and 11 make the horizontal canvas the client window.

Line 14 defines a canvas with vertical split bars.

Lines 16 through 23 set the text for each static control and position the text in each pane.

Figure 40 shows the completed split canvas.

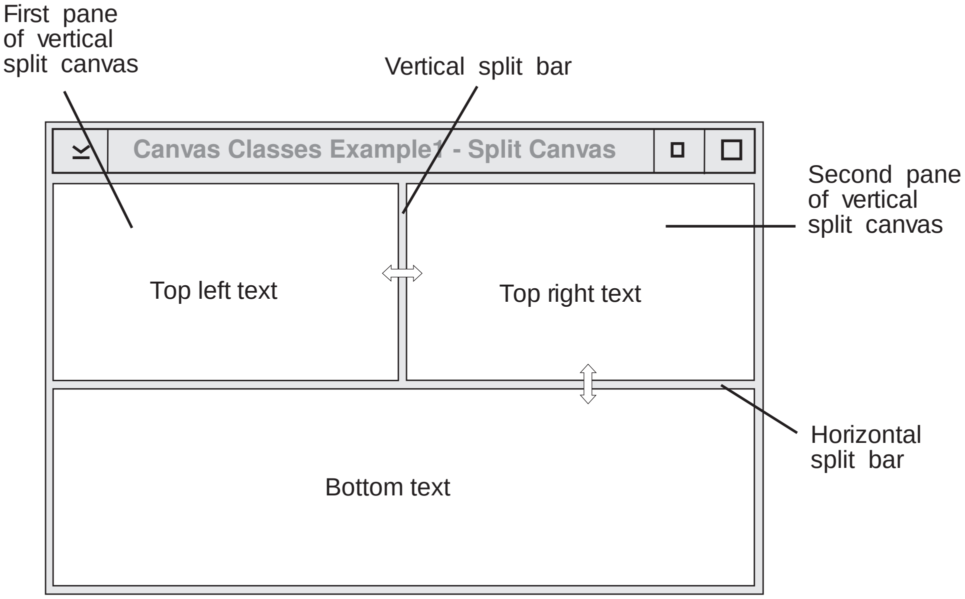

First pane of vertical split canvas
Vertical split bar
Canvas Classes Example1 - Split Canvas
Second pane of vertical split canvas
Top left text
Top right text
Horizontal split bar
Bottom text

Figure 40. Split Canvas Example

Understanding Set Canvases

A *set canvas* arranges its child controls in either rows or columns. The User Interface Class Library uses the term *deck* for either a row or column. You can arrange the decks of a set canvas either horizontally or vertically. The set canvas attempts to place the same number of controls in each deck.

Each deck is created large enough to contain the largest control in the deck. To do this, the canvas calls the minimumSize member function for each child control. For controls that have sizes defined by the text they contain, such as push buttons and radio buttons, this default processing is normally sufficient. However, for a control that does not have a well-defined size, such as a list box or multiple-line edit control, you need to set its minimum size by overriding the calcMinimizeSize member function or by calling its setMinimumSize member function before adding it to the set canvas.

Note: Your application can determine the best minimum size for list boxes, multiple-line edit controls, containers, and frame windows.

The order in which you create the child controls determines their positions on the set canvas and the order in which tab and cursor keys switch focus between the controls. Several styles are available to control the orientation of the decks and the placement of controls within the decks. You can also alter the spacing between controls and between the decks and the edge of the canvas.

Creating a Set Canvas

The following example shows you how to create a set canvas.

1. Declare the classes that use a set canvas as follows:

```
/***********************************************************/
/* Create the command handler                              */
/***********************************************************/
class ACommandHandler : public ICommandHandler {

public:
  ACommandHandler(MyWindow *asetcan);

protected:
  virtual Boolean command(ICommandEvent& cmdEvent);

private:
  MyWindow *setcan_type;
};
⋮
/***********************************************************/
/* Create the frame window                                 */
/***********************************************************/
class MyWindow : public IFrameWindow

   {
   public:
      MyWindow(unsigned long windowId);
      ~MyWindow();
      Boolean command( ICommandEvent& evt );
      MyWindow & updateSetCan(unsigned long eventtype);
   private:
      void cleanUpMemory();
      ITitle myTitle;
      IMenuBar myMenu;
      void statusBox(IString messageText);
      ISetCanvas      *myCanvas;
      IRadioButton    *rbt1;
      IRadioButton    *rbt2;
      IRadioButton    *rbt3;
      IPushButton     *pbt1;
      IPushButton     *pbt2;
      IPushButton     *pbt3;
      IStaticText     *st1;
      IScrollBar      *sb1;
      ICheckBox       *chkbx1;
      ICheckBox       *chkbx2;
      ICheckBox       *chkbx3;
      ACommandHandler * commandHandler;
   };
```

2. Define the class and its children in the .cpp file, as follows:

```
/***********************************************************/
/* Window constructor                                      */
/***********************************************************/
MyWindow::MyWindow(unsigned long windowId) :
   IFrameWindow(windowId),
   myTitle(this, "Set Canvas Example")

{
         myCanvas = new ISetCanvas(ID_CANVAS, this, this);
         rbt1 = new IRadioButton(ID_RB1, myCanvas, myCanvas, IRectangle(),
            IRadioButton::classDefaultStyle);
         rbt2 = new IRadioButton(ID_RB2, myCanvas, myCanvas, IRectangle(),
            IRadioButton::classDefaultStyle);
         rbt3 = new IRadioButton(ID_RB3, myCanvas, myCanvas, IRectangle(),
            IRadioButton::classDefaultStyle);
         chkbx1 = new ICheckBox(ID_CB1, myCanvas, myCanvas, IRectangle(),
            ICheckBox::classDefaultStyle | IControl::group);
         chkbx2 = new ICheckBox(ID_CB2, myCanvas, myCanvas, IRectangle(),
            ICheckBox::classDefaultStyle);
         chkbx3 = new ICheckBox(ID_CB3, myCanvas, myCanvas, IRectangle(),
            ICheckBox::classDefaultStyle);
         pbt1 = new IPushButton(ID_PB1, myCanvas, myCanvas, IRectangle(),
            IPushButton::classDefaultStyle | IPushButton::defaultButton);
         pbt2 = new IPushButton(ID_PB2, myCanvas, myCanvas, IRectangle(),
            IPushButton::classDefaultStyle | IPushButton::defaultButton);
         pbt3 = new IPushButton(ID_PB3, myCanvas, myCanvas, IRectangle(),
            IPushButton::classDefaultStyle | IPushButton::defaultButton);
         rbt1->setText("Button1");
         rbt2->setText("Button2");
         rbt3->setText("Button3");
         chkbx1->setText("CheckBox1");
         chkbx2->setText("CheckBox2");
         chkbx3->setText("CheckBox3");
         pbt1->setText("Pushbutton1");
         pbt2->setText("Pushbutton2");
         pbt3->setText("Pushbutton3");
         myCanvas->setDeckCount(3);
         myCanvas->setDeckOrientation(ISetCanvas::vertical);
         setClient(myCanvas);
         myCanvas->refresh();


//*********************************************************
// Create Command Handler
//*********************************************************
commandHandler = new ACommandHandler(this);
commandHandler->handleEventsFor(this);
}
```

Figure 41 shows a set canvas created by using the preceding example.

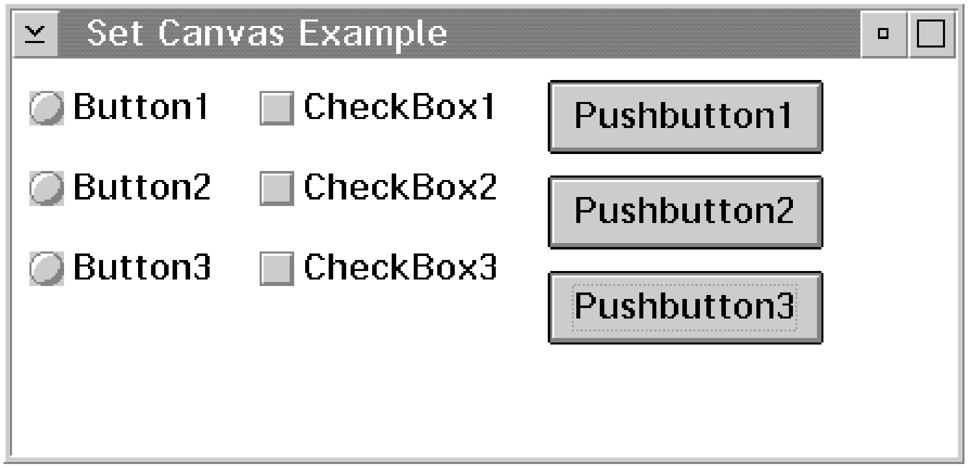


Figure 41. Set Canvas Example

The following examples use a split canvas as a client area. Two set canvases, each with seven radio buttons, are then added to the split canvas.

1. This code from the header file declares the ASetCanvas class as a subclass of IFrameWindow.

```
#include <iframe.hpp>                   // IFrameWindow
#include <istattxt.hpp>                 // IStaticText
#include <iradiobt.hpp>                 // IRadioButton
#include <isetcv.hpp>                   // ISetCanvas
#include <isplitcv.hpp>                 // ISplitCanvas
#define  NUMBER_OF_BUTTONS  14

class ASetCanvas : public IFrameWindow
{
  public:                               //Define the public information
    ASetCanvas(unsigned long windowId); //Constructor for this class
   ~ASetCanvas();                       //Destructor for this class

  private:                              //Define private information
    ISplitCanvas    clientCanvas;
    IStaticText     status;
    ISetCanvas      vSetCanvas,
                    hSetCanvas;
    IRadioButton  * radiobut[NUMBER_OF_BUTTONS];
    AButtonHandler  buttonHandler;
};
```

2. This code from the .CPP creates the window.

```
⋮
ASetCanvas::ASetCanvas(unsigned long windowId)
  : IFrameWindow( windowId )
  , clientCanvas( WND_SPLITCANVAS, this, this, IRectangle(),
                ISplitCanvas::horizontal | IWindow::visible )
```

```
 5   , status(WND_STATUS, &clientCanvas, &clientCanvas)
 6   , vSetCanvas(WND_VSETCANVAS, &clientCanvas, &clientCanvas)
 7   , hSetCanvas(WND_HSETCANVAS, &clientCanvas, &clientCanvas)
 8 {
 9
10   setClient(&clientCanvas);                     //Make split canvas the client area
11
12   status.setAlignment(IStaticText::centerCenter);//Set alignment of status area text
13
14   vSetCanvas.setDeckOrientation(ISetCanvas::vertical);
15   vSetCanvas.setDeckCount(3);                   //Create 3 vertical decks in top canvas
16
17   hSetCanvas.setDeckOrientation(ISetCanvas::horizontal);
18   hSetCanvas.setDeckCount(3);                   //Create 3 horizontal decks in bottom canvas
19   hSetCanvas.setPad(ISize(10,10));              //Set some space around buttons
⋮
```

Lines 6 and 7 create the two set canvases.

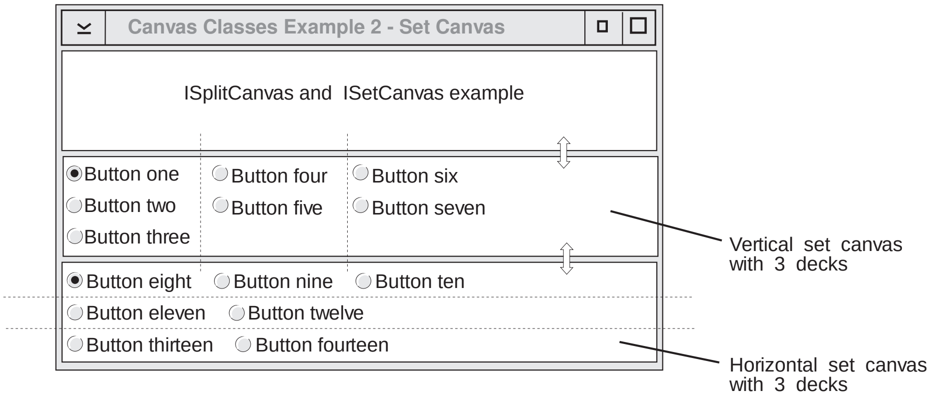
Line 10 makes the split canvas the client area.

Line 12 sets the alignment of the static text control to be centered vertically and horizontally.

Lines 14 and 15 set the deck orientation of the first canvas to vertical and the deck count to 3.

Lines 17 through 19 set the deck orientation of the second canvas to horizontal and the deck count to 3. They also set the padding around the second canvas to 10 pels.

Figure 42 shows the set canvas created using this code.



Vertical set canvas with 3 decks

Horizontal set canvas with 3 decks

Figure 42. Set Canvas Example

Understanding Multiple-Cell Canvas

A *multiple-cell canvas* consists of a grid of rows and columns and looks like a spreadsheet. You place child controls on the canvas by specifying the starting cell and the number of contiguous rows and columns that they can span. You can refer to cells in the grid by the column and row value. The top left cell coordinate is (1,1).

The default cell size is 10 pixels high by 10 pixels wide. The actual number of rows and columns in the canvas is the highest row and column value used. For example, a radio button is placed at (4,5) and a push button at (2,7). Therefore, the canvas has 4 columns and 7 rows. Columns and rows can also be referenced by setColumnWidth and setRowHeight.

The initial size of a row or column is determined by the size of the largest control in that row or column. By default, the row and column sizes are fixed. You can make the rows and columns expandable by using setColumnWidth and setRowHeight (setting the parameter expandable=true), in which case sizing the canvas also sizes them.

Figure 43 shows a multiple-cell canvas. Notice that the small window's location is at column 4, row 5 (4,5), while the large window starts at column 9, row 2 (9,2). The large window is 2 columns wide and 3 rows long.

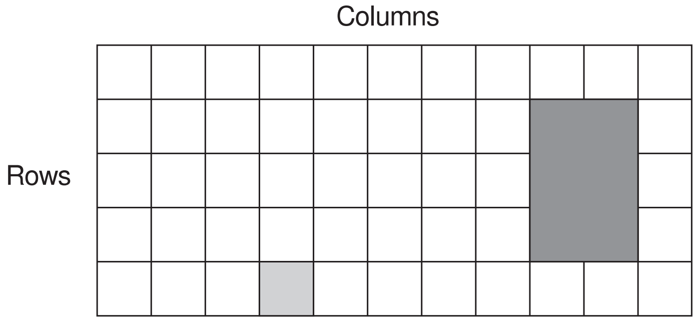


Figure 43. A Multiple-Cell Canvas

Note: A child window in a fixed-sized row and column is automatically sized based on its minimum size, which is typically based on the window's content and the available area within the window.

You can also leave rows and columns empty to provide spacing between child controls. If you do not explicitly size the empty rows and columns using setRowHeight and setRowWidth, they are sized to the default cell size.

Multiple-Cell Canvas

You cannot have more than one window occupy the same “starting cell”. (Starting cell means the start row and column you specify when you call addToCell.) However, you can have overlapping cells.

The size of a row or column is initially established by calculating the minimum size needed to hold all the windows in the row or column. In practice, this means the width of a column is set to the minimum width of the largest window in the column. The width is determined by calling IWindow::minimumSize on all windows in the column. The application can set the width or height of a column or row, respectively. If an attempt is made to set a value less than the minimum size of the largest window, the value is saved, but ignored since the minimum size is the larger of the hard coded size for the cell and the minimum size of the window. In other words, the application can reserve more space than the window needs but can not cause the clipping of the window to occur.

Refer to the *Open Class Library Reference* for more information about IWindow::minimumSize.

A column or row never has a smaller width or height than the minimum size of the largest window in the column or row. If the windows later calculate a smaller minimum size, the width or height of the column or row is the larger of the following:

- The new minimum size of the largest window in the column or row
- The column width or row height set by the application

The minimum size of the canvas is based on the minimum sizes of all the windows on the canvas plus the size of empty rows and columns. If the area allotted to the canvas is less than this size, the canvas is clipped at the lower right corner. You can use the IViewPort class to add scroll bars to handle this situation.

See “Understanding View Ports” on page 377 and the *Open Class Library Reference* for more information about the IViewPort class.

The multiple-cell canvas has two styles that can help you create layouts with a multiple-cell canvas by providing a way to easily visualize the placement and size of each child window. The gridLines style causes the multiple-cell canvas to have grid lines between the rows and columns of the canvas. Grid lines are stationary. If you want grid lines that you can move with the mouse, use the dragLines style, which causes the multiple-cell canvas to have draggable grid lines between the rows and columns of the canvas.

Here are some considerations when using grid lines and drag lines:

- Grid lines and drag lines are placed at the left and top edges of cells.

- Child windows can overwrite grid lines and drag lines.
- The use of grid lines or drag lines does not change the initial placement or sizing of child windows.
- Moving a drag line causes the child windows on each side of the drag line to be resized in a manner similar to that of an ISplitCanvas.

  🕮 Refer to the *Open Class Library Reference* for more information about ISplitCanvas.

Creating a Multiple-Cell Canvas

The following example shows you how to create a multiple-cell canvas.

1. Declare a class that uses a multiple-cell canvas control in the .hpp file with static text and entry field controls.

```
/**********************************************************/
/* MyWindow class declaration                             */
/**********************************************************/
class MyWindow : public IFrameWindow

{
public:
      MyWindow(unsigned long windowID);
      ~MyWindow();

protected:
      Boolean command(ICommandEvent& cmdEvent);

private:
      IMultiCellCanvas      myClient;
      IStaticText           stName,
                            stAddress,
                            stId,
                            stBirthday,
                            stPhone;
      IEntryField           efName,
                            efAddress,
                            efId,
                            efBirthday,
                            efPhone;
      ACommandHandler      * commandHandler;
};
```

2. Construct the multiple-cell canvas in the .cpp file.

```
MyWindow::MyWindow( unsigned long windowId)
  : IFrameWindow("MultiCell Canvas Example",windowId,
                 IFrameWindow::defaultStyle(),
    myClient( ID_MCC, this, this),
    stName( ID_ST_NAME,  &myClient, &myClient ),
    stAddress( ID_ST_ADDR,  &myClient,&myClient ),
    stId( ID_ST_ID,   &myClient, &myClient ),
    stBirthday( ID_ST_BD,   &myClient, &myClient ),
    stPhone( ID_ST_PHONE, &myClient, &myClient ),
    efName( ID_EF_NAME,  &myClient, &myClient ),
    efAddress( ID_EF_ADDR,  &myClient, &myClient ),
    efId( ID_EF_ID,    &myClient, &myClient ),
    efBirthday( ID_EF_BD,    &myClient, &myClient ),
    efPhone( ID_EF_PHONE, &myClient, &myClient )
  {

   setClient( &myClient );

//*********************************************************
// Build multiple-cell canvas with fields
//*********************************************************
  myClient.addToCell( &stName, 2, 2 );
  myClient.addToCell( &efName, 3, 2 );
  myClient.addToCell( &stAddress, 2, 3 );
  myClient.addToCell( &efAddress, 3, 3 );
  myClient.addToCell( &stId, 2, 4 );
  myClient.addToCell( &efId, 3, 4 );
  myClient.addToCell( &stBirthday, 2, 5 );
  myClient.addToCell( &efBirthday, 3, 5 );
  myClient.addToCell( &stPhone, 2, 6 );
  myClient.addToCell( &efPhone, 3, 6 );

//*********************************************************
// Load the static text fields
//*********************************************************
  stName.setText( "Name:" );
  stName.setForegroundColor(IColor::yellow);
  stName.setAlignment( IStaticText::centerRight );
  stAddress.setText( "Address:" );
  stAddress.setForegroundColor(IColor::yellow);
  stAddress.setAlignment( IStaticText::centerRight );
  stBirthday.setText( "Birthdate:" );
  stBirthday.setForegroundColor(IColor::yellow);
  stBirthday.setAlignment( IStaticText::centerRight );
  stId.setText( "ID:");
  stId.setForegroundColor(IColor::yellow);
  stId.setAlignment( IStaticText::centerRight );
  stPhone.setText( "Phone:");
  stPhone.setForegroundColor(IColor::yellow);
  stPhone.setAlignment( IStaticText::centerRight );

}
```

The following examples show you how to create a window containing a multiple-cell canvas. The canvas contains two check boxes, two radio buttons, three static text controls, and one push button.

1. This code from the header file declares the AMultiCellCanvas class as a subclass of IFrameWindow class.

```
#include <iframe.hpp>                         // IFrameWindow
#include <istattxt.hpp>                       // IStaticText
#include <ipushbut.hpp>                       // IPushButton
#include <iradiobt.hpp>                       // IRadioButton
#include <icheckbx.hpp>                       // ICheckBox
#include <imcelcv.hpp>                        // IMultiCellCanvas
```

```
class AMultiCellCanvas : public IFrameWindow
{
  public:
    AMultiCellCanvas(unsigned long windowId);

  private:
    IMultiCellCanvas    clientCanvas;
    IStaticText         status,
                        title1,
                        title2;
    ICheckBox           check1,
                        check2;
    IRadioButton        radio1,
                        radio2;
    IPushButton         pushButton;
};
```

2. This code creates the window.

```
AMultiCellCanvas::AMultiCellCanvas(unsigned long windowId)
  : IFrameWindow(windowId)
  , clientCanvas( WND_MCCANVAS, this, this )
  , status( WND_STATUS, &clientCanvas, &clientCanvas )
  , title1( WND_TITLE1, &clientCanvas, &clientCanvas )
  , title2( WND_TITLE2, &clientCanvas, &clientCanvas )
  , check1( WND_CHECK1, &clientCanvas, &clientCanvas )
  , check2( WND_CHECK2, &clientCanvas, &clientCanvas )
  , radio1( WND_RADIO1, &clientCanvas, &clientCanvas )
  , radio2( WND_RADIO2, &clientCanvas, &clientCanvas )
  , pushButton( WND_PUSHBUT, &clientCanvas, &clientCanvas )
{

  setClient( &clientCanvas );      // make multicell canvas the client area
  status.setAlignment( IStaticText::centerCenter );// set status area text
  status.setText( STR_STATUS );

  title1.setAlignment( IStaticText::centerLeft );    // set text and alignment
  title1.setText( STR_TITLE1 );

  title2.setAlignment( IStaticText::centerLeft );    // set text and alignment
  title2.setText( STR_TITLE2 );

  check1.setText( STR_CHECK1 );                  // set check box text
  check2.setText( STR_CHECK2 );
  radio1.setText( STR_RADIO1 );                  // set radio button text
  radio2.setText( STR_RADIO2 );

  pushButton.setText( STR_PUSHBUT );

  radio2.select();                       // preselect one radio button
  check1.enableGroup().enableTabStop();// set tabStop and group styles
  radio1.enableGroup().enableTabStop();
  pushButton.enableGroup().enableTabStop();

  clientCanvas.addToCell(&status  , 1, 1, 4, 1);     // add controls to canvas.
  clientCanvas.addToCell(&title1  , 1, 3, 2, 1);     // the canvas runs from
  clientCanvas.addToCell(&title2  , 3, 3, 2, 1);     // 1,1 to 4,7
  clientCanvas.addToCell(&check1  , 2, 4);           // only one row and
  clientCanvas.addToCell(&check2  , 2, 5);           // one column are
41   clientCanvas.addToCell(&radio1  , 4, 4);           // expandable, as this
42   clientCanvas.addToCell(&radio2  , 4, 5);           // allows the canvas to
43   clientCanvas.addToCell(&pushButton , 2, 7);        // fill the client area.
44
45   clientCanvas.setRowHeight(2, 20, true); // make row 2 20 pixels high and expandable
46
47   clientCanvas.setRowHeight(6, 40); // make row 6 40 pixels high
48
49   clientCanvas.setColumnWidth(4, 40, true);  // make column 4 40 pixels wide and expandable
50
51   check1.setFocus();                         // set focus to first check box
52   show();                                    // show main window
53
54 } /* end AMultiCellCanvas :: AMultiCellCanvas(...) */
```

Note: There is not a 1:1 relationship between OS/2 and AIX pixels.

Line 3 creates a multiple-cell canvas.

Line 14 makes it the client area.

Lines 36 through 43 place the other controls on the canvas using the addToCell member function.

Lines 45 through 49 set the sizes for rows 2 and 6 and column 4. Row 2 and column 4 are expandable.

Figure 44 shows the completed multiple-cell canvas.

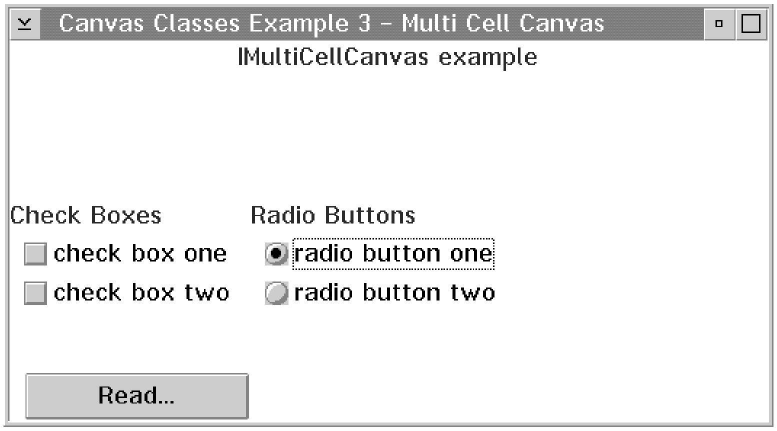


Figure 44. Multiple-Cell Canvas Example with 4 Columns and 7 Rows

Understanding View Ports

A *view port* canvas provides a scrollable view area with horizontal and vertical scroll bars. By default, the scroll bars display only when needed. A view port can have only one child control. The size of the child control is fixed. If the view port is smaller than the child control, the view port allows the user to scroll the child control.

If you need more than one control in a view port, place the controls into another type of canvas, which you can then make the child of the view port.

Creating a View Port

The following example shows how to create a view port.

1. Declare a view port in the .hpp file.

```
/***********************************************************/
/* AViewWindow declaration                                 */
/***********************************************************/
class AViewWindow : public IFrameWindow {

 public:
    AViewWindow(unsigned long windowId);
    ~AViewWindow();
    AViewWindow & moveHoriz();
    AViewWindow & moveVert();
 private:
    ITitle          title;
    IViewPort       viewPort;
    ICanvas         canvas;
    IStaticText     text;
    IMultiLineEdit  mle1;
    IMultiLineEdit  mle2;
    ACommandHandler *commandHandler;
};
```

2. Define the view port in the .cpp file as shown in the following code:

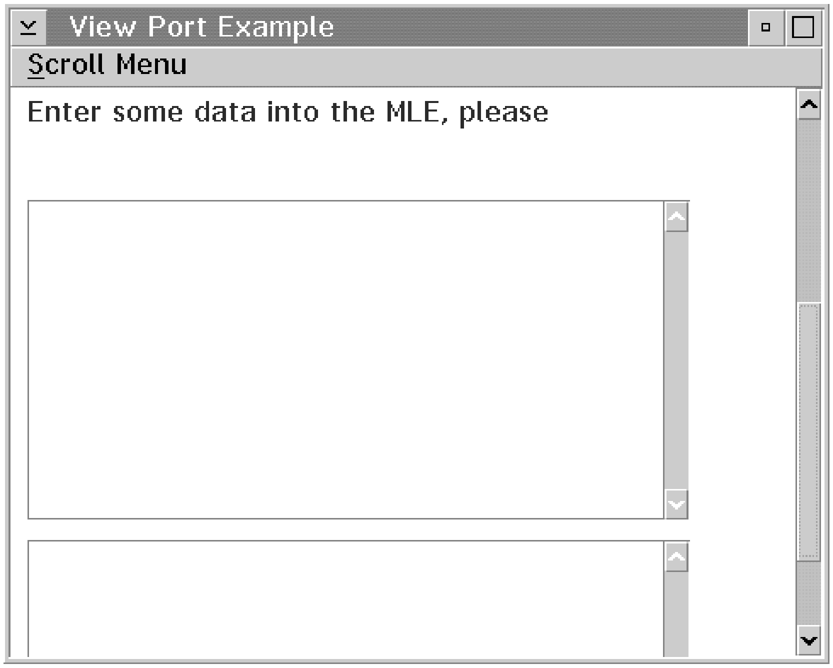
```
/*************************************************************/
/* Window constructor                                        */
/*************************************************************/
AViewWindow::AViewWindow(unsigned long windowId)
   : IFrameWindow(windowId,
     IFrameWindow::defaultStyle() |
     IFrameWindow::menuBar),
     title(this, "View Port Example"),
     viewPort(WID_VIEWPORT, this,this, IRectangle()),
     canvas(WID_CANVAS, &viewPort, &viewPort, IRectangle(5, 5, 410, 460)),
     text(WID_TEXT, &canvas, &canvas, IRectangle(10, 360, 400, 410)),
     mle1(WID_MLE1, &canvas, &canvas, IRectangle(10, 0, 400, 149)),
     mle2(WID_MLE2, &canvas, &canvas, IRectangle(10, 160, 400, 349))

{

  viewPort.setColor(IViewPort::fillBackground, IColor::yellow);
  setClient(&viewPort);
  text.setText("Enter some data into the MLE, please");

//*************************************************************
// Command Handler
//*************************************************************
```

```
  commandHandler = new ACommandHandler(this);
  commandHandler->handleEventsFor(this);
  sizeTo(ISize(500,300));
  show();

}

AViewWindow::~AViewWindow()
{
  commandHandler->stopHandlingEventsFor(this);
  delete commandHandler;
}
```

3. Handle events, as follows:

```
/************************************************************/
/* Horizontal move event handler                            */
/************************************************************/
AViewWindow & AViewWindow :: moveHoriz()
 {
      viewPort.scrollViewHorizontallyTo(200);
      return(*this);
 }

/************************************************************/
/* Vertical move event handler                              */
/************************************************************/
AViewWindow & AViewWindow :: moveVert()
 {
      viewPort.scrollViewVerticallyTo(200);
      return(*this);
 }

/************************************************************/
/* Command handler                                          */
/************************************************************/
ACommandHandler::ACommandHandler(AViewWindow *viewWindow)
{
  view = viewWindow;
}

/************************************************************/
/* Command handler                                          */
/************************************************************/
IBase::Boolean ACommandHandler::command(ICommandEvent & cmdEvent)
{
  Boolean eventProcessed(true);

  switch (cmdEvent.commandId())
  {
    case ID_HORIZ:
      view->moveHoriz();
      break;
    case ID_VERT:
      view->moveVert();
      break;
    default:
     eventProcessed = false;
  }
 return true;
}
```

Figure 45 shows the view port created with the preceding code.



Enter some data into the MLE, please

Figure 45. A View Port

View Port

32

Creating and Using File and Font Dialogs

A *dialog box* is a special, short-lived window that you use to display information and receive input from the user in a structured dialog format. The information is typically related to a particular action being performed by the application.

If you develop portable applications, use a canvas for dialog windows.

Refer to Chapter 49, “Adding Dialogs and Push Buttons” on page 621 for more information about using canvases for dialogs.

This chapter covers file and font dialogs.

Specifying File Dialog Information

The *file dialog* enables a user to specify a file to be opened or a file name in which current work is to be saved. It includes the function to switch directories and logical drives. The IFileDialog class lets you define the dialog for files. To create a file dialog, follow these steps:

1. Set the file dialog using the optional feature of the IFileDialog class to specify initial settings for the dialog you create. To use this feature, create an instance of the Settings class when you create the dialog, as shown in the following example:

   ```
   IFileDialog::Settings fsettings;
   ```

   The Settings class has several member functions, including:

   - setFileName
   - setOpenDialog
   - setPosition
   - setSaveAsDialog

   Note: The setOpenDialog is the default. If you want a **Save As** dialog, use the setSaveAsDialog member function.

   To set the dialog, use the following statements:

   ```
   fsettings.setTitle(STR_FILEDLGT);     //Set open dialog title from resource
   fsettings.setFileName("*.hlo");       //Set file names to *.hlo
   ```

2. Create an instance of the IFileDialog class after setting up the dialog. Use the following statements:

   ```
   IFileDialog * fd=new IFileDialog(     //Create file open dialog
      desktopWindow(),                   //Parent is desktop
      this,                              //Owner is me
      fsettings);                        //  with settings
   ```

Refer to the *Open Class Library Reference* for other ways to define an instance of the IFileDialog class.

3. Test the response from the file dialog using the pressedOK member function. This member function returns true if the user ended the dialog by pressing **OK**.

4. Read the resulting file name from the file dialog. Use the fileName member function to return the fully qualified name that the user selected.

For the complete sample code, see the openFile member function in the AHELLOW6.CPP file (Hello World version 6). The Hello World samples, are in the `\ibmcpp\samples\ioc` directory.

Creating a File Dialog

The following example demonstrates creating both an **Open** file dialog and a **Save As** file dialog.

1. Define the main window and a command handler to handle menu command events.

```
class MyCommandHandler : public ICommandHandler
{
  public:
     MyCommandHandler( MyWindow* theOwner );
     virtual
        ~MyCommandHandler();
     virtual Boolean
        command( ICommandEvent& event );

  private:
     MyWindow
       *owner,
       *parent;
};

class MyWindow : public IFrameWindow
{
   public:
      MyWindow();

   private:
       ITitle title;
       IMenuBar menu;
       MyCommandHandler cmds;
};
```

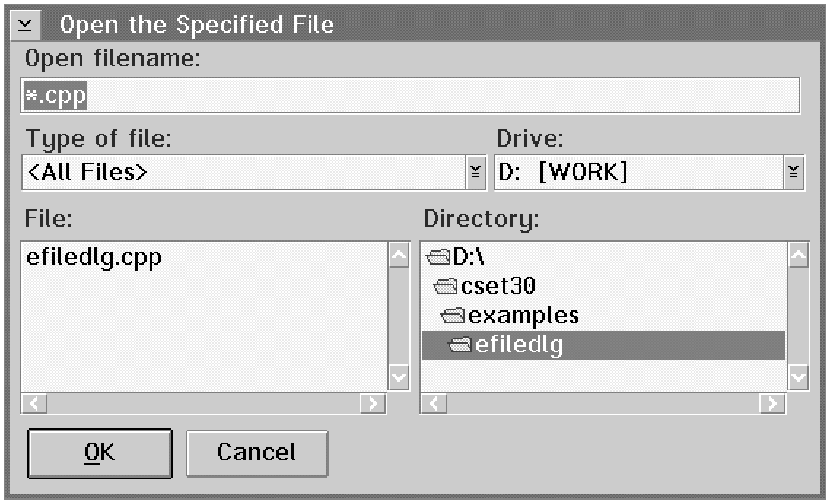
2. Create the main window.

```
MyWindow::MyWindow() :
          IFrameWindow( ID_MAIN ),
          title( this, "File Dialog Example" ),
          menu( ID_MENU, this ),
          cmds( this )
{
  cmds.handleEventsFor( this );
  setFocus().show();
}
```

3. Create the file dialogs based on the menu command events.

```
⋮
IBase::Boolean MyCommandHandler::command( ICommandEvent& event )
{
  Boolean
    rv = false;
  IMessageBox
    msgbox( owner );

  switch ( event.commandId() )
  {
    case ID_FILE_OPEN :
    {
      IFileDialog::Settings fileSettings;
      fileSettings.setTitle( "Open the Specified File" ); // Set open dialog title
      fileSettings.setFileName( "*.cpp" );                // Set file names to *.cpp
      IFileDialog filedlg( parent, owner, fileSettings );
      if ( filedlg.pressedOK() )
      {
        msgbox.show( IString( "You selected the file '" )
                     + IString( filedlg.fileName() )
                     + IString( "'." ),
                     IMessageBox::okButton );
      }
      break;
    }

    case ID_FILE_SAVEAS :
    {
      IFileDialog::Settings fileSettings;
      fileSettings.setTitle( "Save to the Specified File" );  // Set save-as dialog title
      fileSettings.setFileName( "noname.txt" );              // Set file name to noname.txt
      fileSettings.setSaveAsDialog();
      IFileDialog filedlg( parent, owner, fileSettings );
      if ( filedlg.pressedOK() )
      {
        msgbox.show( IString( "You saved the file '" )
                     + IString( filedlg.fileName() )
                     + IString( "'." ),
                     IMessageBox::okButton );
      }
      break;
    }
  }
  return rv;
}
```

Figure 46 shows the file dialog created using the preceding example.



Open the Specified File

Open filename: *.cpp

Type of file: <All Files>

Drive: D: [WORK]

File: efiledlg.cpp

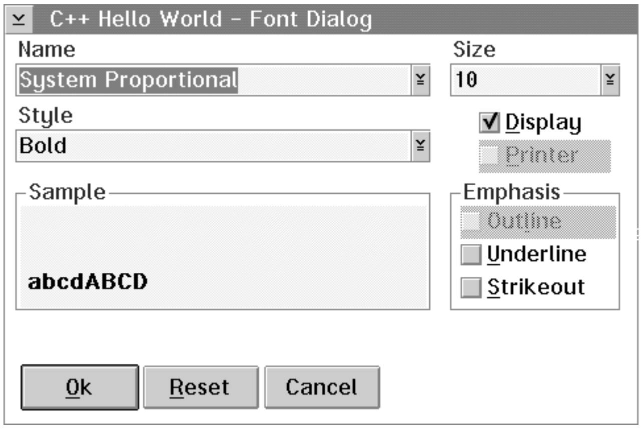
Directory: D:\ cset30 examples efiledlg

Figure 46. File Dialog Example

Specifying Font Dialog Information

The *font dialog* enables a user to specify a choice of font names, styles, and sizes from the range of those available in a given application. Use the IFontDialog class to handle fonts in your applications.

Figure 47 shows an example of a font dialog.



C++ Hello World - Font Dialog

Name: System Proportional

Size: 10

Style: Bold

☑ Display

□ Printer

Sample: abcdABCD

Emphasis

□ Outline

□ Underline

□ Strikeout

Figure 47. Example of a Font Dialog

Creating a Font Dialog

The following section describes how to create a font dialog, using the Hello World version 6 application. The following code comes from the AHELLOW6.CPP file.

```
AHelloWindow &
  AHelloWindow :: setHelloFont()
{
⋮
  IFont tempFont(&hello);
  IFontDialog::Settings fontSettings(&tempFont);
  fontSettings.setTitle(IResourceId(STR_FONTDLGT));
  IFontDialog fontDialog( desktopWindow(), this,
                          IFontDialog::resetButton, fontSettings);
  if (fontDialog.pressedOK())
  {
    hello.setFont(tempFont);
  }
  return (*this);                       //Return a reference to the frame
⋮
}; /* end AHelloWindow :: setHelloFont() */
```

In the preceding sample, the font in an IStaticText control is changed to the font the user selects from an IFontDialog. This is done by:

1. Creating an IFont object called tempFont that represents the font currently being used by the IStaticText control pointed to by hello.
2. Passing a pointer to the tempFont object on the constructor to an IFontDialog::Settings object called fsettings.
3. Passing the fsettings object on the IFontDialog constructor.

Because fontSettings is constructed using tempFont, the IFontDialog initially displays the name, style, size, and emphasis associated with tempFont (for example, the font currently used by the IStaticText object). If the user dismisses the IFontDialog by pressing **OK**, then tempFont automatically updates itself to reflect the font the user chose via the IFontDialog. The setFont member function can be used to actually change the font of the IStaticText control to tempFont.

Refer to Chapter 39, “Understanding Fonts” on page 487 to see how to set a font.

Font Dialog

33 Creating Menus

A *menu* is a window that contains a list of items—text strings, bit maps, or images drawn by the application—that enables the user to choose from these predetermined choices using a mouse or keyboard. The types of menus are the menu bar, pull-down menu, cascaded menu, and pop-up menu.

A menu is always owned by another window, usually a frame window. When a user makes a selection from a menu, the command handler gets the menu selection event.

Creating Menu Bars and Pull-Down Submenus

A typical application uses a menu bar and several pull-down submenus. The pull-down submenus ordinarily are hidden, but they become visible when the user makes selections from the menu bar. You can map pull-down submenus from the menu bar, another pull-down menu, or a pop-up menu.

The menu bar is a child of the frame window; the menu bar window handle is the key to communicating with the menu bar and its submenus.

PM CUA suggests that when you have more than three levels of submenus, you create a dialog.

Motif Motif suggests that when you have more than three cascaded menus, you create a dialog.

Understanding Pop-Up Menus

A *pop-up menu* is a menu that is displayed when a user presses the appropriate key or mouse button. A pop-up menu contains choices that can be applied to an object at the time the menu is displayed.

The User Interface Class Library provides the IPopUpMenu and IMenuHandler classes to manipulate pop-up menus. To display a pop-up menu in your application, subclass IMenuHandler, override the makePopUpMenu function, and construct an IPopUpMenu object.

Creating Pop-Up Menus

This section shows you two ways to create a pop-up menu.

Version 6 of the Hello World application creates two pop-up menus: one for the “Hello, World” static text control and one for the Earth window static text control. The contents of the pop-up menus are defined in the AHELLOW6.RC resource file as follows:

```
⋮
MENU WND_HELLOPOPUP
  BEGIN
    MENUITEM "~Left-align text",  MI_LEFT
    MENUITEM "~Center text"    ,  MI_CENTER
    MENUITEM "~Right-align text", MI_RIGHT
  END
MENU WND_EARTHPOPUP
  BEGIN
    MENUITEM "~Twinkling stars", MI_TWINKLE
    MENUITEM SEPARATOR
    MENUITEM "~Brighten stars",  MI_BRIGHT
    MENUITEM "~Dim stars",       MI_DIM
  END
⋮
```

In the AHELLOW6.HPP file, an APopUpHandler class is defined to process requests for making the pop-up menus appear.

```
⋮
class APopUpHandler : public IMenuHandler {
protected:
virtual Boolean
  makePopUpMenu(IMenuEvent& menuEvent);
};
⋮
```

The makePopUpMenu member function is called whenever the user requests a pop-up menu, usually by clicking mouse button 2, in a window for which menu requests are being handled. This function shows the appropriate pop-up menu. You can also dynamically create the pop-up menu in the makePopUpMenu function.

Hello World version 6 demonstrates how to create a pop-up menu as a data member of the AHelloWindow class and how to dynamically create a pop-up menu. Typically, you choose one of these approaches based on resource balancing. Static pop-up menus are only created and deleted once, but take up storage whether they are needed or not; dynamic pop-up menus are created and deleted on demand, which can slow processing if you request them frequently.

The following sample is from the AHELLOW6.CPP file:

```
IBase::Boolean
  APopUpHandler::makePopUpMenu(IMenuEvent &menuEvent)
{
  Boolean eventProcessed(true);         //Assume event will be processed
  IPopUpMenu     *popUpMenu;
/*--------------------- Create Pointer to Owner Window --------------------|
|  The window pointer stored in the menu event points to the owner of      |
|    the menu to be popped up.                                             |
|------------------------------------------------------------------------*/
  IWindow        *popUpOwner = menuEvent.window();
  IStaticText    *hello;
  AEarthWindow   *earth;
/*--------------------- Select the Pop-up Menu ----------------------------|
|  Determine which menu to pop up based on the window ID of the window     |
|    for which the menu event is being handled.                            |
|------------------------------------------------------------------------*/
  switch (popUpOwner->id()) {
    case WND_HELLO:
/*--------------------- Setup the Hello Window Pop-up ---------------------|
|  If the hello window pop-up is requested, then get a pointer to the      |
|    pop-up using windowWithId.  This pop-up was created as a data member  |
|    of the class that contains the owner window to demonstrate how to     |
|    pop up previously-created, static, pop-up windows.                    |
|  The pointer, hello, is created by casting the popUpOwner to the type    |
|    of the AHelloWindow::hello object.  The pointer calls
|    AHelloWindow::alignment is called to determine
|    which pop-up meun item to disable.                                    |
|------------------------------------------------------------------------*/
      popUpMenu = (IPopUpMenu *)
            IWindow::windowWithId(WND_HELLOPOPUP,popUpOwner);
      if (popUpMenu)
      {
        hello = (IStaticText *)popUpOwner;
        popUpMenu->enableItem(MI_LEFT,
                        hello->alignment()!=IStaticText::centerLeft);
        popUpMenu->enableItem(MI_CENTER,
                        hello->alignment()!=IStaticText::centerCenter);
        popUpMenu->enableItem(MI_RIGHT,
                        hello->alignment()!=IStaticText::centerRight);
      }
                                        //If the pop-up wasn't found,
      else eventProcessed=false;        //  the event wasn't processed
      break;
    case WND_EARTH:
/*--------------------- Setup the Earth Window Pop-up ---------------------|
|  If the earthWindow pop-up is requested, then create the pop-up menu     |
|    dynamically, with the earthWindow as the owner.  This approach is     |
|    used to create pop-up menus on demand.  The setAutoDeleteObject       |
|    function is used to automatically delete the menu after the user      |
|    has made a selection.                                                 |
|  The pointer, earth, is created by casting the popUpOwner to the type    |
|    of the earthWindow object.  The pointer is used to call the           |
|    AEarthWindow  isTwinkling and isBright functions.                     |
|  The Twinkling menu item contains a check mark when isTwinkling is true.|
|  Either the Bright or Dim menu item is disabled, depending on the        |
|    result of isBright.                                                   |
|------------------------------------------------------------------------*/
```

```
   popUpMenu = new IPopUpMenu(WND_EARTHPOPUP,popUpOwner);
   if (popUpMenu)
   {
     popUpMenu->setAutoDeleteObject();
     earth = (AEarthWindow *)popUpOwner;
     popUpMenu->checkItem(MI_TWINKLE,earth->isTwinkling());
     if (earth->isBright())
     { popUpMenu->disableItem(MI_BRIGHT); }
     else
     { popUpMenu->disableItem(MI_DIM); }
   }
                                    //If the pop-up wasn't created,
   else eventProcessed=false;       //  the event wasn't processed
   break;
  default:                          //If the owner wasn't hello or earth,
   eventProcessed=false;            //  the event wasn't processed
 } /* end switch */
/*----------------------- Show the Pop-up Menu ---------------------------|
|  If the pop-up menus were setup successfully, use                       |
|    IPopUpMenu::show to show the menu at the position        |
|    where the pointing device was when the menu event occurred.          |
|-----------------------------------------------------------------------*/
 if (eventProcessed)
   popUpMenu->show(menuEvent.mousePosition());
 return(eventProcessed);
} /* end APopUpHandler :: makePopUpMenu(...) */
```

The case statement for WND_HELLO uses the IWindow::windowWithId function to get a pointer to the static pop-up menu. The WND_EARTH case creates a pop-up menu dynamically. Because Hello World version 6 creates the menu using the new operator, it must also be deleted. The easiest way to delete a dynamic pop-up menu is to use the setAutoDeleteObject function, which causes the menu to be automatically deleted when the menu ends.

In either case, when the menu is found or created, the makePopUpMenu function displays the menu by using IPopUpMenu::show. The mouse pointer position, which is taken from the menu event, is passed as one argument to specify where the pop-up menu will appear.

The makePopUpMenu function is only called for windows that are attached to the pop-up menu handler. Hello World version 6 attaches a pop-up menu handler directly to the IStaticText objects, hello and earthWindow. It attaches to the objects instead of the frame because static text objects do not pass events up the owner chain. Therefore, you should use this approach if you develop portable applications. For the same reason, the command handler attaches to the static text objects so that the command events that result from using the pop-up menus will be sent to the command handler. In Hello World version 6, this is done using the following code from the AHELLOW6.CPP file.

```
⋮
commandHandler.handleEventsFor(&hello);
commandHandler.handleEventsFor(&earthWindow);
popUpHandler.handleEventsFor(&hello);
popUpHandler.handleEventsFor(&earthWindow);
⋮
```

By reusing the existing command handler, commands such as MI_LEFT can have the same processing whether they are generated by a menu bar item, a push button, an accelerator key, or a pop-up menu.

The following example also shows you how to create a pop-up menu. To create a pop-up menu for a list box, follow the steps in this section.

1. Declare a handler which is a subclass of both IMenuHandler and ICommandHandler so that the same handler can be used for handling the pop-up menu and the commands that originate from it. Make an instance of this handler a child of the frame window. Make the pop-up menu itself a dynamic child of the handler.

```
/**********************************************************/
/*  Define the frame window                               */
/**********************************************************/
class AppWindow : public IFrameWindow {

 public:
    AppWindow(unsigned long windowId);
    ~AppWindow();

 private:
    ITitle            title;
    IMenuBar          menu;
    IMultiCellCanvas  canvas;
    IStaticText       sttxt;
    IListBox          listbox;
    PopUpHandler      * pLBPopUp;
    ACommandHandler   * commandHandler;
};

/**********************************************************/
/*  Define the pop-up and command handler                 */
/**********************************************************/
class PopUpHandler : public IMenuHandler,
                     public ICommandHandler
{
    public:
      PopUpHandler(IListBox & lbUpdate, IStaticText & stMsg);
      ~PopUpHandler();

    protected:
      virtual Boolean makePopUpMenu(IMenuEvent& menuEvent);
      virtual Boolean command(ICommandEvent& cmdevt);

⋮
```

```
    private:
      void setLBColor(unsigned long ulNewColor);
      void setLBText(unsigned long ulNewSize);
      IListBox * pLB;
      IStaticText * pST;
      IPopUpMenu  * pPopUpMenu;
      unsigned long ulColor;
      unsigned long ulText;
};
```

2. Create an instance of the pop-up handler and start handling events in the frame window constructor, as shown in the following code:

```
/***********************************************************/
/* Construct the frame window with children                */
/***********************************************************/
AppWindow::AppWindow(unsigned long windowId)
   : IFrameWindow(windowId, defaultStyle()),
     title(this, "PopUp","Example"),
     canvas(ID_CANVAS, this, this),
     listbox(ID_LISTBX, &canvas, &canvas),
     sttxt(ID_TEXT, &canvas, &canvas),
     menu(ID_MENU, this),
     pLBPopUp(0)
{
 listbox.setForegroundColor(IColor::yellow);
 sttxt.setText("ListBox Set to Default Values");
 sttxt.setLimit(30);
//**************************************************
// Customize the multiple-cell canvas
//**************************************************
setClient(&canvas);
canvas.addToCell(&listbox, 2, 2);
canvas.setRowHeight(2, 20, true);
canvas.setColumnWidth(2, 20, true);
canvas.addToCell(&sttxt, 2, 6);
canvas.setRowHeight(1, 10);
canvas.setRowHeight(4, 25);
canvas.setRowHeight(7, 10);
canvas.setColumnWidth(1, 10);
canvas.setColumnWidth(3, 10);

// Attach the pop-up menu handler to the list box
pLBPopUp = new PopUpHandler(listbox, sttxt);

// Handle commands from the menu bar
commandHandler = new ACommandHandler(this);
commandHandler->handleEventsFor(this);

}
```

3. Create the pop-up menu and start event handling in the PopUpHandler constructor. Show the pop-up menu in the makePopUpMenu function. Handle the events generated from the pop-up menu in the command function, as follows:

```
/**********************************************************/
/* Pop-up handler constructor                             */
/**********************************************************/
PopUpHandler::PopUpHandler(IListBox & LBUpdate, IStaticText & stMsg)
 : pLB(0),
   pST(0),
   pPopUpMenu(0),
   ulColor(0),
   ulText(0)
{
   unsigned long ulColor, ulText;
⋮

 // Need pointers to entry field and static text to update them
 // from the exit
 pLB=  &LBUpdate;
 pST = &stMsg;

 // Create the pop-up menu
 pPopUpMenu = new IPopUpMenu(ID_POPUP2, pLB);

 // Default color and alignment values
 ulColor = ID_BLUE;
 ulText  = ID_ADD_FIRST;

 // Check the default items
 pPopUpMenu->checkItem(ulColor);
 pPopUpMenu->checkItem(ulText);

 // Set default color and alignment
 setLBColor(ulColor);
 setLBText(ulText);

 // Handle menu events and pop-up menu requests
 ICommandHandler::handleEventsFor(pLB);
 IMenuHandler::handleEventsFor(pLB);
}

⋮

/**********************************************************/
/* Handle the menu functions to change the list box color  */
/**********************************************************/
void PopUpHandler::setLBColor(unsigned long ulNewColor)
{
   switch (ulNewColor) {
   case ID_RED:
      pLB->setBackgroundColor(IColor::red);
      pST->setText("Changed color to Red");
      break;
   case ID_GREEN:
      pLB->setBackgroundColor(IColor::green);
      pST->setText("Changed color to Green");
      break;
   case ID_BLUE:
      pLB->setBackgroundColor(IColor::blue);
      pST->setText("Changed color to Blue");
      break;
```

```
    default:
      break;
   } /* endswitch */
}
⋮

/*********************************************************/
/* Handle the menu functions to add text to the list box   */
/*********************************************************/
void PopUpHandler::setLBText(unsigned long ulNewText)
{
   switch (ulNewText) {
   case ID_ADD_FIRST:
        pLB->addAscending("I need to learn C++");
        pST->setText("Added text to list box in ascending order");
      break;
   case ID_DELETE_TEXT:
        pLB->removeAll();
        pST->setText("Deleted All Text from ListBox");
      break;
   case ID_ADD_LAST:
        pLB->addDescending("I know C++");
        pST->setText("Added text to list box in descending order");
      break;
   default:
      break;
   } /* endswitch */
}

/*********************************************************/
/* Show the pop-up menu created in the constructor         */
/*********************************************************/
IBase::Boolean PopUpHandler::makePopUpMenu(IMenuEvent& menuEvt)
{
   pPopUpMenu->show(menuEvt.mousePosition());
   return true;
}

⋮

/*********************************************************/
/* Command handling for the pop-up menu                    */
/*********************************************************/
IBase::Boolean PopUpHandler::command(ICommandEvent& cmdevt)
{
   switch (cmdevt.commandId())
   {
      case ID_RED:
      case ID_GREEN:
      case ID_BLUE:
        setLBColor(cmdevt.commandId());
        return true;

      case ID_ADD_FIRST:
      case ID_DELETE_TEXT:
        case ID_ADD_LAST:
          setLBText(cmdevt.commandId());
          return true;
      }
      return false;
   }
```

System Menu

The system menu in the upper-left corner of a standard frame window is different from the menus defined by the application. The system menu is controlled and defined by the system. However, using User Interface Class Library, you can enable and disable items on the system menu, add or delete items, or even decide not to include the system menu on your frame window. The ISystemMenu class gives you access to the system menu and allows you to manipulate its items. The IFrameWindow::Style class provides the flag systemMenu which allows you to specify the presence of the system menu on your frame window.

Refer to *Open Class Library Reference* for more information on these classes.

System Menu

34 Creating and Using Notebooks

A *notebook* control is a visual component that organizes related information on individual “pages” so that a user can find and display that information. It simulates a real notebook and provides the user with a recognizable visual component. It consists of a notebook client area, a binding, page buttons, optional tabs, and an optional status area. Users select and display pages using a mouse or the keyboard.

You can customize a notebook to meet varying application requirements, while providing a user-interface component that can be used easily to develop products that conform to the Common User Access (CUA) user-interface guidelines. Your application can specify different colors, sizes, and orientations for its notebooks, but the underlying function of the control remains the same.

Use the INotebook class objects to create notebooks or wrapper existing notebooks. These INotebook objects function like other control objects in the library. You can use a notebook as the client of a frame window, outside the client area of a frame extension, or within any canvas class. Typically, you create a notebook as the client of a frame window.

Understanding the Default Notebook Styles

You can specify notebook styles during notebook creation to define the look and feel of the notebook, or use the default notebook styles provided by the User Interface Class Library. The default notebook styles are the following:

INotebook::backPagesBottomRight
Simulates recessed pages along the right and bottom edges of the notebook.

INotebook::majorTabsRight
Places major tabs on the right side of the notebook.

INotebook::solidBinding
Binding is solid.

INotebook::squareTabs
Tabs are square-shaped.

INotebook::statusTextLeft
Status text is left-justified.

INotebook::tabTextCenter
Tab text is centered.

Default Notebook Styles

IWindow::visible
Notebook is visible.

Figure 48 shows the appearance of a notebook control created using the default notebook styles. This notebook is a modified version of the notebook sample contained in the `\ibmcpp\samples\ioc\notebook` directory

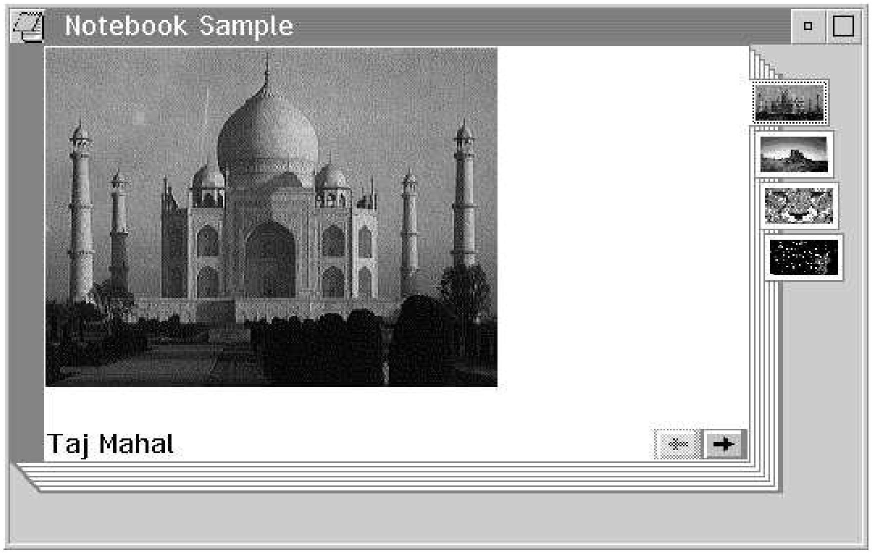


Figure 48. Notebook Control Created with Default Styles

The notebook control shown in Figure 48 resembles a real notebook in its general appearance. For example, the notebook has a *binding* that, along with recessed pages on the right and bottom edges, gives the notebook a three-dimensional appearance. The binding is solid and placed on the left side, using the default INotebook styles backPagesBottomRight and solidBinding.

In the bottom right corner of the notebook are the *page buttons*. These buttons are for bringing one page of the notebook into view at a time. They are a standard component provided with every notebook. Your application can change the default width and height of the page buttons using INotebook::setPageButtonSize. However, you cannot query the size of the page buttons.

Selecting the *forward page button* (the arrow pointing to the right) causes the next page to be displayed; selecting the *backward page button* (the arrow pointing to the left) causes the previous page to be displayed.

Default Notebook Styles

To the left of the page buttons when using the default notebook style is the *status line*, which enables your application to provide information to the user about the page currently displayed. The notebook does not supply any default text for the status line. You are responsible for using INotebook::PageSettings::setStatusText or INotebook::setStatusText to associate a text string with the status line of each page. By default, the text in the status line is left-justified.

The page buttons are always located in the corner where the recessed edges of the notebook intersect. These recessed edges are called the *back pages*. The default notebook's back pages intersect in the bottom right corner, which means the recessed pages are on the bottom and right edges.

The back pages are important because their intersection determines where the *major tabs* can be placed, which in turn determines the placement of the binding and the *minor tabs*. You can use major and minor tabs to organize related pages into sections; minor tabs define subsections within major tab sections. The content of each section has a common theme, which is represented to the user by a tabbed divider, similar to a tabbed page in a notebook.

The default style, INotebook::majorTabsRight, specifies that major tabs, if used, are placed on the right side of the notebook. This is the default placement when the back pages intersect at the bottom right corner of the notebook. The binding is located on the left because it is always located on the opposite side of the notebook from the major tabs.

Minor tabs are always placed perpendicular to the major tabs, based on the intersection of the back pages and the major tab placement.

Note: You can only specify one major or minor tab attribute for each notebook page in OS/2 Version 2.x This restriction is removed in OS/2 Warp Version 3.0.

Minor tabs are displayed only as the associated major tab page is selected or if the notebook has no major tab pages.

The default shape of the tabs used on notebook divider pages is square. You can change the default width and height of the major and minor tabs using INotebook::setMajorTabSize and INotebook::setMinorTabSize, respectively.

A notebook tab may contain either text or a bitmap. You can place text on a tab using INotebook::PageSettings::setTabText or INotebook::setTabText. Use INotebook::PageSettings::setTabBitmap or INotebook::setTabBitmap to place a bitmap on a tab. You cannot position a bitmap on a tab using the default support because the bitmap stretches to fill the rectangular area of the tab. However, you may use the owner draw support to control the positioning and drawing of the bitmap on a tab.

Creating a Notebook

Use the INotebook class to create and manage the notebook control window. You can create an object of this class using one of the following constructors:

```
INotebook( unsigned long     windowId,
           IWindow*          parent,
           IWindow*          owner,
           const IRectangle& initial = IRectangle(),
           const Style&      style = defaultStyle() );

INotebook( unsigned long windowId,
           IWindow*      parentAndOwner );

INotebook( const IWindowHandle& handle );
```

Only the first of the three constructors creates a new notebook control. This constructor accepts a numeric identifier for the notebook, a pointer to a window object for its parent window, and a pointer to a window object for its owner window. You can optionally specify the position, size, and styles of the notebook.

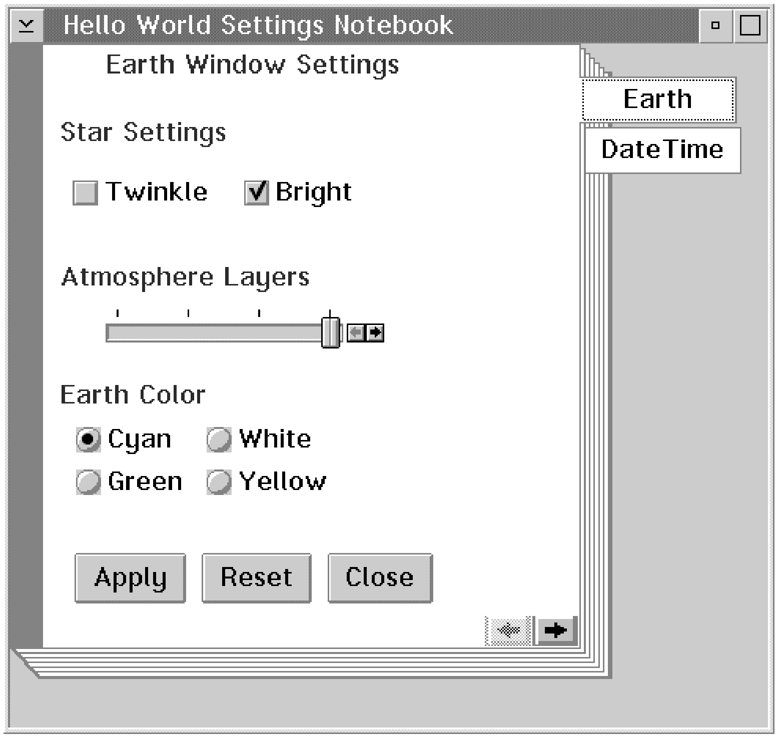
The next two constructors wrapper an existing notebook control. The first of these two constructors is designed to wrapper a notebook control that is loaded as a dialog resource. It accepts a numeric identifier for the notebook, and a pointer to a window object for its parent and owner window. The last of these two constructors is designed to wrapper an existing notebook control. It accepts the existing notebook's window handle.

Version 6 of the Hello World application, hello6, creates a notebook as a private data member, helloSettingsNotebook, from a derived IFrameWindow class, called ANotebookWindow. The helloSettingsNotebook object is initialized on the ANotebookWindow constructor using the following INotebook constructor from the ANOTEBW6.CPP file:

```
⋮
,helloSettingsNotebook(WND_NOTEBOOK, this, this)
⋮
```

This constructor creates the notebook as a child window of the ANotebookWindow object and uses the default style.

Figure 49 shows the Hello World version 6 notebook control.



Hello World Settings Notebook

Earth Window Settings

Star Settings

□ Twinkle ☑ Bright

Atmosphere Layers

Earth Color

● Cyan ○ White
○ Green ○ Yellow

Figure 49. Hello World Version 6 Settings Notebook

Specifying Notebook Styles

You can specify notebook styles during notebook creation to define the look and feel of the notebook. The User Interface Class Library provides notebook styles so that your application can specify or change the notebook's styles.

See INotebook::Style in the *Open Class Library Reference* for a complete list of available notebook styles.

Note: When you specify an INotebook::Style on the notebook constructor, ensure that no conflicts occur. Many of the style choices are not independent from one another.

If you specify more than one style bit, you must use a bitwise OR operator (|) to combine them.

Refer to “Combining Styles” on page 315 for more information about bitwise operators.

Notebook Styles

If you want to specify notebook styles other than the default when you are creating a notebook, create an object of the INotebook::Style class, initialize it, and pass a reference to it on the constructor that accepts a style parameter. For example:

```
INotebook::Style
  style = INotebook::spiralBinding |
          INotebook::roundedTabs;
```

The notebook created using the preceding statements has a spiral binding and tabs with rounded corners.

A modified version of the Hello World notebook is shown in Figure 50.

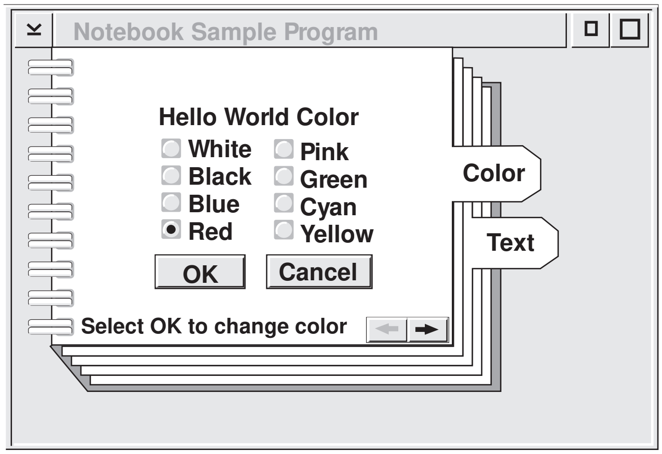


Figure 50. Notebook Control

ANOTEBW6.CPP file:

```
⋮
INotebook::PageSettings
      helloSettings(INotebook::PageSettings::majorTab|
                    INotebook::PageSettings::autoPageSize);
helloSettings.setTabText(IResourceId(STR_EARTHTAB));
helloSettingsNotebook.addFirstPage(helloSettings,&earthPage);
helloSettings.setTabText(IResourceId(STR_DATETIMETAB));
helloSettingsNotebook.addLastPage(helloSettings,&dateTimePage);
#ifndef IC_MOTIF
helloSettingsNotebook.setMajorTabSize(ISize(100,30));
#endif
⋮
```

Each notebook page window, in this case, is a multiple-cell canvas. Hello World version 6 creates a PageSettings object, helloSettings, to contain the specifications for each notebook page. The object is created with the INotebook::PageSettings::Attributes. that specify major tabs and automatic page sizing. The tab text is set to the text used to label the earthPage using the resource ID of the string to load from the resource file.

Hello World version 6 creates the first notebook page by specifying the INotebook::PageSettings object and the page window object to be associated with this page. To create the next page, load the notebook tab text for the dateTimePage from the resource file into the page settings and use INotebook::addLastPage. If you wanted to add more pages to Hello World version, you would use INotebook::addLastPage to append pages to the end of the notebook.

Removing Notebook Pages

You can remove pages from the notebook by supplying the IPageHandle that was returned when the page was created.

Use the following INotebook functions to remove notebook pages:

removePage
Accepts a handle directly or determines the handle from an instance of an INotebook::Cursor.

removeAllPages
Removes multiple pages of a notebook.

removeTabSection
Removes the pages associated with a major or minor tab section.

See the *Open Class Library Reference* for more information about these INotebook functions.

Changing Notebook Colors

Your application can tailor the color of almost any part of the notebook. Use the various color functions to change the colors of a notebook.

When you change the color in a control area, Presentation Manager passes this color change request to all the children of the control. This causes a child window with the same color control area to change to the new specified color if its control area has not been explicitly set. Therefore, changing colors in the notebook can cause changes to the page windows on the notebook.

Use the following INotebook functions to change the notebook's color:

setPageBackgroundColor
setMajorTabBackgroundColor
setMinorTabBackgroundColor
setMajorTabForegroundColor
setMinorTabForegroundColor

Notebook Styles

setBackgroundColor
setForegroundColor

See the *Open Class Library Reference* for information about the supported color functions in INotebook.

35 Creating and Using Containers

A *container* control holds objects. OS/2 provides a variety of containers, such as folders, templates, and the Workplace Shell itself. Containers can display their objects in different views: tree, icon, text, name, and details views. Using the User Interface Class Library, you can also develop your own containers and change their views, behaviors, and layouts.

Figure 51 shows an example of a container.

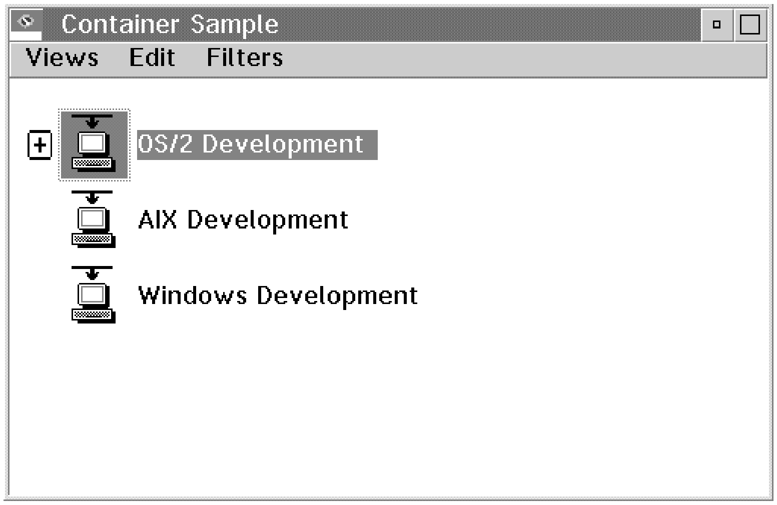


Figure 51. Example of a Container

Understanding Containers

Containers are defined by the Common User Access (CUA) architecture.

Use the IContainerControl class to create an instance of a container object. With this class, you can control, for example, the view of the objects inside the container. The following example shows one way to create a container:

```
IContainerControl cnrCtl(CNR_RESID, this, this);
```

Several styles are available for containers that you can use to manage such activities as multiple-selection and automatic positioning.

You can define the styles in the constructor, or you can use member functions to set the style required after you create an instance of the container object. An example of a style statement is highlighted in the following:

```
cnrCtl = new IContainerControl (CNR_RESID, this, this);
cnrCtl->setExtendedSelection();
```

Refer to the *Open Class Library Reference* to learn about other styles and related member functions.

Creating Container Objects

Because a container has no meaning without its objects, use the IContainerObject class to create objects to put into it. At a minimum, an IContainerObject has an icon and a name.

The following is an example of an IContainerObject constructor:

```
IContainerObject  ( const IString&          string,
                    const IPointerHandle&   iconHandle = 0);
```

To design your own objects for your applications, create a class that is derived from the IContainerObject class. If you use multiple inheritance, you must list the IContainerObject class first. To create a container object with department names, addresses, and zip codes for your company, define this class as follows:

```
class Department : public IContainerObject
{
   public:
        Department(const IString& Name,
                   const IPointerHandle& Icon,
                   const IString& Code,
                   const IString& Address);

        IString Code()
          const {  return strCode; }

        IString Address()
          const {  return strAddress; }

        void setCode (IString code)
          {strCode = code;}

        void setAddress (IString address)
          {strAddress = address;}

        virtual void handleOpen
          (IContainerControl* container);

   private:

      IString strAddress;
      IString strCode;
};
```

The statements for a constructor definition are:

```
Department :: Department(const IString& Name,
                         const IPointerHandle& Icon,
                         const IString& Code,
                         const IString& Address):
       IContainerObject(Name, Icon),
       strCode (Code),
       strAddress (Address),
       {}
```

After you define the class, create an instance of an object using the following statement:

```
dept1 = new Department (
             "OS2 Development",
             IApplication::current().userResourceLibrary().loadIcon(IBMLOGO),
             "TWPD",
             "Building 71");
```

Adding and Removing Container Objects

After you create the objects and the container, add the objects to the container.

The following statements add objects to the container, cnrCtl. The first line adds an object, dept1. The next three lines add dept2, dept3, and dept4 in a hierarchy under dept1. The last line adds dept5.

```
cnrCtl->addObject(dept1);       // Add Department 1 to container
cnrCtl->addObject(dept2,dept1); // Add Department 2 under Department 1
cnrCtl->addObject(dept3,dept1); // Add Department 3 under Department 1
cnrCtl->addObject(dept4,dept1); // Add Department 4 under Department 1
cnrCtl->addObject(dept5);       // Add Department 5 to container
cnrCtl->addObject(dept6);       // Add Department 6 to container
```

When you place the container in the client window and show the window and the container, you see a window like the one in Figure 52.

Container Objects

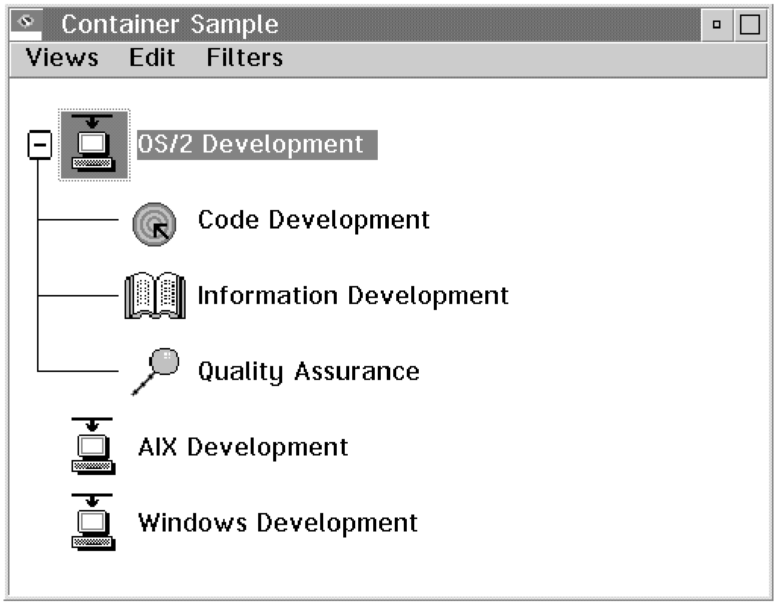


Figure 52. Example of a Container Showing Objects in an Expanded Tree View

The window shows a tree view of the container’s objects. This view is discussed later.

You can also use the ICnrAllocator class to allocate a list of container items to be inserted into an IContainerControl. When you construct instances of this class, you can allocate memory from the container control for a specified number of objects with one call.

The IContainerControl::addObjects member function inserts all the initialized items of the allocator with one call to the container.

The following example shows how to use the ICnrAllocator class and the IContainerControl::addObjects member function:

```
/*********************************************/
/* Define your derived IContainerObject class */
/*********************************************/
class MyObject : public IContainerObject
{
public:
  MyObject(const IString& name) : IContainerObject(name) {}
  ~MyObject() {}
};

/*********************************/
/* Create a frame and a container */
```

```
/**********************************/
IFrameWindow frame(0x1300);
IContainerControl cnr(0x1400, &frame, &frame);
cnrCtl.showTextView();

/*************************************************/
/* Create an allocator and allocate 10000 objects */
/*************************************************/
ICnrAllocator allocator(10000, sizeof(MyObject));

/***********************/
/* Initialize all 10000 */
/***********************/
for(int i=0; i<10000; i++)
{
  new(allocator) MyObject("Peter");
}

/***************************************/
/* Add all the objects to the container */
/***************************************/
cnrCtl.addObjects(allocator);
```

By default, the container only removes objects when the container is deleted. It does not delete them. However, you can delete all objects in the container when the container is deleted by using the following code statement:

```
cnrCtl->setDeleteObjectsOnClose();
```

You can call IContainerControl::deleteAllObjects to delete all objects in a container. Specify the style IContainerControl::noSharedObjects when you create a container that does not share any objects with other containers. This increases the performance of the IContainerControl::deleteAllObjects member function.

The following example shows how to create a container with the noSharedObjects style:

```
/***************************************************/
/* Create a container with the noSharedObjects style */
/***************************************************/
IContainerControl* cnrCtl = new IContainerControl(0x1400, &frame,
                                &frame, IRectangle(0,0,0,0),
                                IContainerControl::defaultStyle() |
                                IContainerControl::noSharedObjects);
```

Sharing Objects Among Containers

You can also create objects and place them in multiple containers. The same object is then shared by two or more different containers.

In Figure 52 on page 408 we have a container with six departments. In the example, dept2, dept3, and dept4 are in a hierarchy under dept1. We now want to create another container with only the main departments. This new container will then share dept1, dept5, and dept6 with the container displayed in Figure 51.

Container Objects

The following statements add three objects to a container, cnrCtl2, that already exist in another container, cnrCtl.

```
/*************************************************/
/*       Container with all departments        */
/*************************************************/
cnrCtl->addObject(dept1);       // Add Department 1 to container
cnrCtl->addObject(dept2,dept1); // Add Department 2 under Department 1
cnrCtl->addObject(dept3,dept1); // Add Department 3 under Department 1
cnrCtl->addObject(dept4,dept1); // Add Department 4 under Department 1
cnrCtl->addObject(dept5);       // Add Department 5 to container
cnrCtl->addObject(dept6);       // Add Department 6 to container
/*************************************************/
/*      Container with main departments only     */
/*************************************************/
cnrCtl2->addObject(dept1);       // Add Department 1 to second container
cnrCtl2->addObject(dept5);       // Add Department 5 to second container
cnrCtl2->addObject(dept6);       // Add Department 6 to second container
```

Since the same object can exist in more than one container, the attributes of an object also reflect the state of that object. For example, an object can be visible in one container but hidden in another. You should consider the state of an object's attribute in each container and the state of the attribute in each place the object resides.

The following container attributes can be modified:

- Visibility
- Cursored emphasis
- Selection emphasis
- In-use emphasis
- Refresh status
- Open status
- Direct edit status
- Expanded or collapsed state in tree view
- Target emphasis
- Source emphasis

Use the base container handler, ICnrHandler, to capture the event notifications provided by the container class. When the values of object attributes in the container change, these series of notifications are captured by the handler and routed to virtual functions within the handler.

For example, for both containers to reflect the same selection emphasis, you must attach an ICnrHandler to keep the objects in the same state in each container. Once both icons are selected, the same action is performed on both containers.

If you are performing multiple actions that cause the container to refresh, you can manipulate the refresh state so that the container will not repaint:

```
cnrCntl.setRefreshOff();
⋮
cnrCntl.setRefreshOn();
cnrCntl.refresh();
```

Filtering Container Objects

You can filter objects in a container. The container uses the FilterFn nested class to show a subset of the existing objects by filtering some of the objects.

To create a filter:

1. Define a class derived from the FilterFn class and override the member function isMemberOf to code the conditions of a valid object.

   The following example defines a FilterFn class:

   ```
   class SelectedObjectsFilter : public IContainerControl::FilterFn
   {
   virtual Boolean
     isMemberOf( IContainerObject* object,
                 IContainerControl* container) const
     {
       return container->isSelected(object);
     }
   };
   ```

   If true is returned by the FilterFn subclass, the container object remains displayed in the container; if false, the object is hidden.

   The isSelected member function returns true if the object has selection emphasis.

   📖 Refer to the *Open Class Library Reference* for information about the types of emphasis.

2. Call IContainerControl::filter. Use the following statements:

   ```
   SelectedObjectsFilter selObjects;
   cnrCtl->filter(selObjects);
   ```

Figure 53 shows how the container appears before you apply the filter.

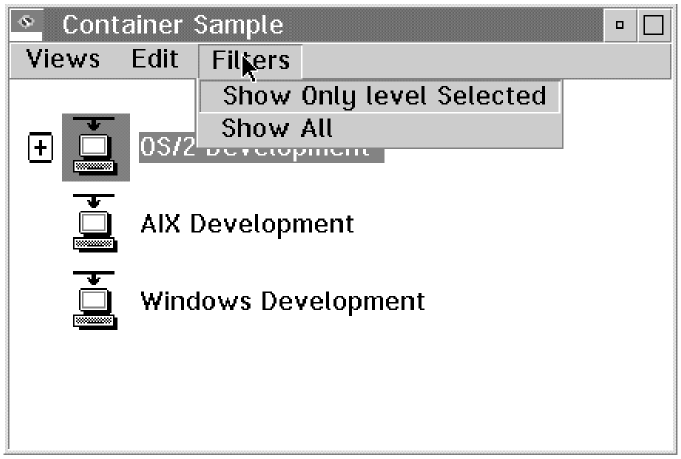


Figure 53. Before Filtering the Container Objects

Figure 54 shows how the container appears after you apply the filter.

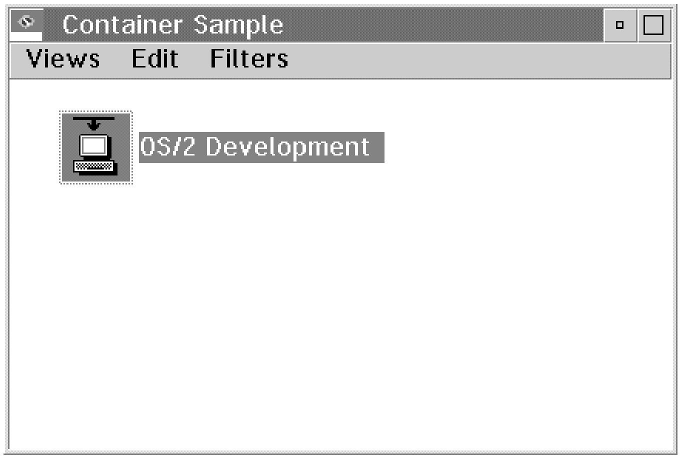


Figure 54. After Filtering the Container Objects

Accessing Container Objects Using an Object Cursor

Use an object cursor to find *all* objects or find *only* those objects that meet a specific criteria.

The following example creates an ObjectCursor and uses it to select all container objects:

```
IContainerControl::ObjectCursor CO1 (*cnrCtl);

for (CO1.setToFirst(); CO1.isValid(); CO1.setToNext())
  {
  cnrCtl->setSelected(cnrCtl->objectAt(CO1));
  }
```

Figure 55 shows the before and after result of setting the selection emphasis.

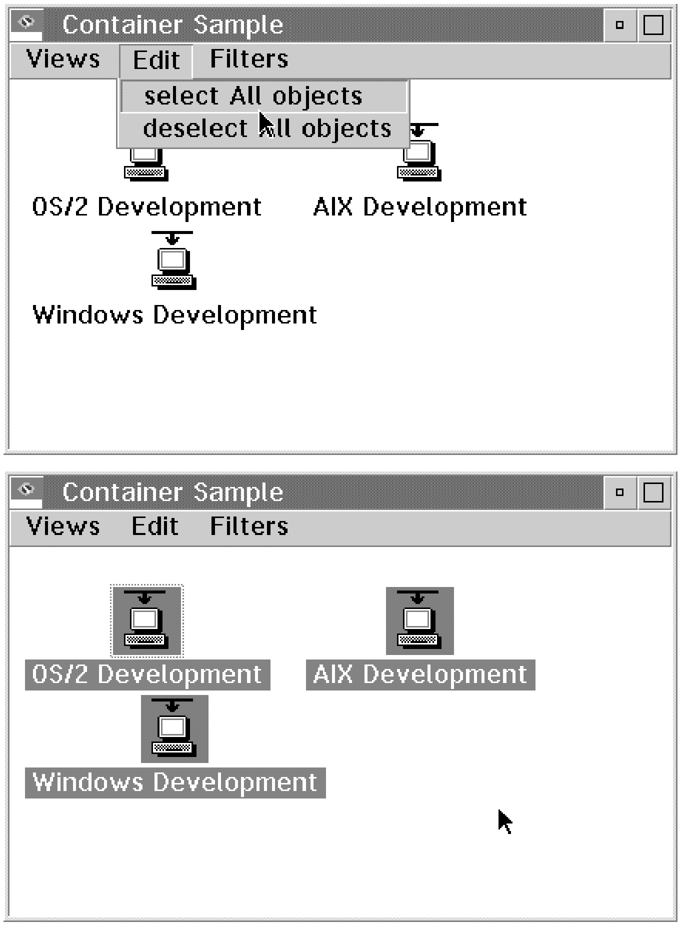


Figure 55. Example of Using an Object Cursor

Changing Views in a Container

You can specify the view using a style on the constructor or set it with a member function. For example, the following statement uses the member function that causes a container to display the icon view:

```
cnrCtl->showIconView();
```

This statement provides the container view shown in Figure 56.

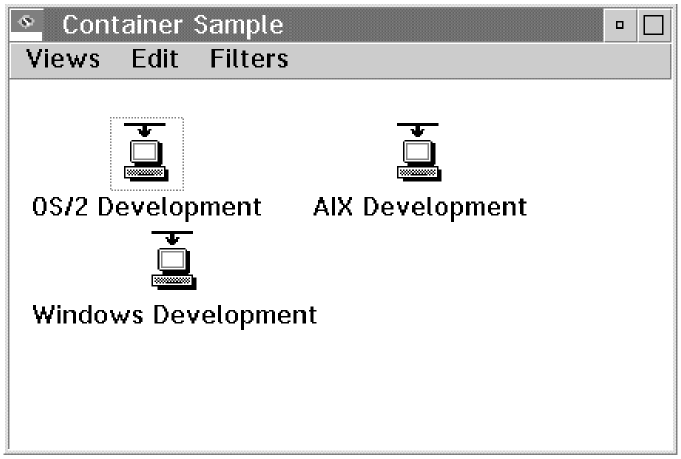


Figure 56. Example of the Icon View

The following statement provides the tree icon view:

```
cnrCtl->showTreeIconView();
```

Figure 57 shows a container with the tree icon view.

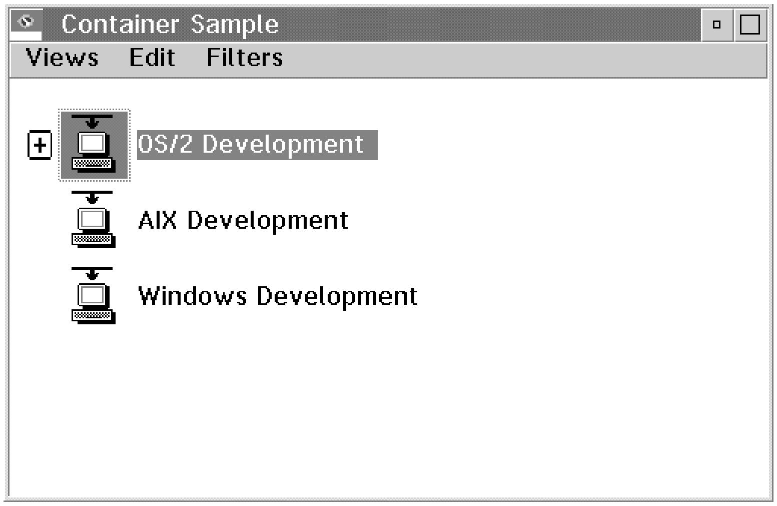


Figure 57. Example of a Tree Icon View

Defining the Details View Using Container Columns

The IContainerColumn class defines the information that is displayed for a given object when the container is in the details view. Only the items that you added with no parent display in the details view. You can use this class to set text in the heading of the columns, add horizontal and vertical separators by column, and align the column contents.

One way to create an instance of an IContainerColumn is for you to provide the offset of the object data to be displayed in the column and, optionally, the styles to be used for the heading and data.

The following shows the syntax for the IContainerColumn constructor:

```
IContainerColumn     ( unsigned long      dataOffset,
                       const HeadingStyle& title = defaultHeadingStyle(),
                       const DataStyle&    data = defaultDataStyle());
```

To create an instance of a container column, use the following statements:

```
colIcon = new IContainerColumn (IContainerColumn::isIcon);

colName = new IContainerColumn (IContainerColumn::isIconViewText);

colCode = new IContainerColumn (offsetof(Department, strCode));

colAddress = new IContainerColumn (offsetof(Department, strAddress));
```

Details View

In the previous example, colIcon, colName, colCode, and colAddress are defined as members of an IFrameWindow. The statements look like this:

```
private:                              //Define private information
  IContainerControl * cnrCtl;
  Department *dept1, *dept2, *dept3, *dept4, *dept5, *dept6 ;
  IContainerColumn *colIcon, *colName, *colCode, *colAddress;
  IMenuBar       * menuBar;
```

After creating the container columns, you can add heading text to them using the following statements:

```
colIcon->setHeadingText("Icon");
colName->setHeadingText("Department Name");
colCode->setHeadingText("Code");
colAddress->setHeadingText("Address");
```

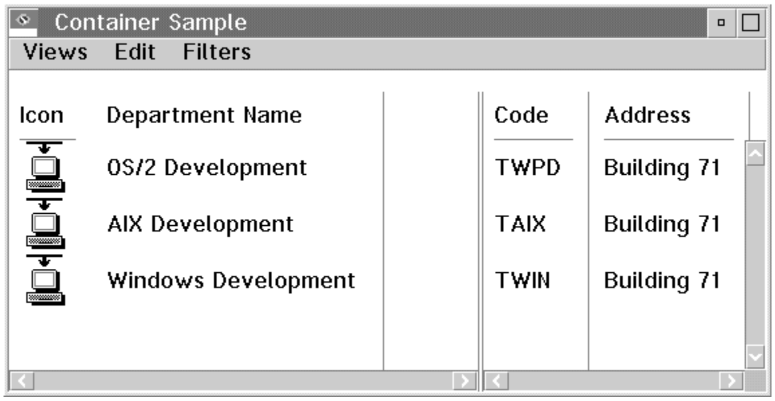
Use showSeparators to add a vertical separator next to a column or a horizontal separator under the heading text. The default adds both. To create only one of the separators, specify it in the member function statement. The following statements show examples of how to create separators:

```
  //Only Horizontal Separator
colIcon->showSeparators(IContainerColumn::horizontalSeparator);
  //Only Vertical Separator
colName->showSeparators(IContainerColumn::verticalSeparator);
colCode->showSeparators(); //both separator by default
colAddress->showSeparators(); //both separator by default
```

After you create the container columns, add them into the container using the following statements:

```
cnrCtl->addColumn(colIcon);
cnrCtl->addColumn(colName);
cnrCtl->addColumn(colCode);
cnrCtl->addColumn(colAddress);
```

Figure 58 is an example of a details view of a container.



Icon	Department Name	Code	Address
	OS/2 Development	TWPD	Building 71
	AIX Development	TAIX	Building 71
	Windows Development	TWIN	Building 71

Figure 58. Example of the Details View

Use the following code statement to put a split bar in the details view by specifying the last column to be viewed in the left window and the location of the split bar in pixels.

```
cnrCtl->setDetailsViewSplit(colName, 350);
```

Creating a Pop-Up Menu in a Container

To create a pop-up menu in a container, create a subclass of ICnrMenuHandler and override the makePopUpMenu. The following statements create the class:

```
class ACnrMenuHandler: public ICnrMenuHandler
{
  public:
  ACnrMenuHandler
    &setCnr(IContainerControl * pcnr)  { pcnrCtl = pcnr; return *this;};

  protected:
    Boolean makePopUpMenu(IMenuEvent& cnEvt);

  private:
    IContainerControl * pcnrCtl;
```

After overriding the makePopUpMenu member function, you can add your own statements. The following statements create a pop-up menu displayed next to a container object with source emphasis:

```
IBase::Boolean ACnrMenuHandler :: makePopUpMenu(IMenuEvent& cnEvt) //
{                                        //
  IPopUpMenu * popUp;                    //Define pop-up variable
  if (popupMenuObject()) {
    popUp=new IPopUpMenu(ID_POPMENU,     //Create pop-up Menu
              cnEvt.window());           //
    if (!pcnrCtl->isDetailsView())       //Details View is only way to edit
    {                                    //  Name, Code and Address so
```

```
      popUp->disableItem(MI_EDNAME);    //  Disable these items if not
      popUp->disableItem(MI_EDCODE);    //  details view.
      popUp->disableItem(MI_EDADDRESS); //
    }
    else
    {
      popUp->disableItem(MI_EDRECORD);  //
    }
    popUp->setAutoDeleteObject();       //
    popUp->show(cnEvt.mousePosition()); //Show Pop-up menu
    pcnrCtl->showSourceEmphasis(popupMenuObject());
    pcnrCtl->setCursor(popupMenuObject());
    return true;                        //Return Pop-up menu
  }
  return false;
};
```

Figure 59 shows the pop-up menu in a container object.

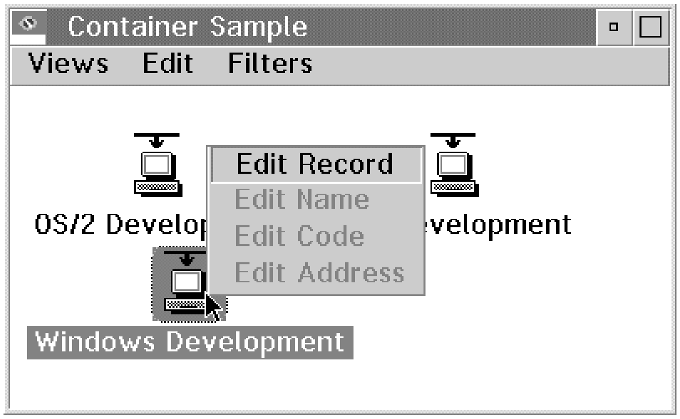


Figure 59. Example of a Pop-Up Menu in a Container Object

Direct Manipulation

36

Supporting Direct Manipulation

Direct manipulation is a user interface technique that lets a user start application functions by manipulating objects. The user begins an action by moving the mouse pointer over an object and then pressing and holding down a mouse button (mouse button 2 is the default) while *dragging* the selected object to a new location. The user then *drops* the object onto the new location by releasing the mouse button. For this reason, direct manipulation is also known as *drag and drop*.

Thus, the user can perform operations directly on objects that appear on the desktop or within an application.

Direct manipulation is not limited to objects in containers, as the object can be a text string in an entry field. Also, users can drag and drop an object onto a new location in the current window, onto another object in a window, or onto a different window.

Direct manipulation is used to move, copy, and link objects. Generally, move is the default operation and is accomplished through the use of the mouse button defined for drag and drop. The other operations, copy and link, are initiated by a user through the use of augmentation keys: the Ctrl (control) key or a combination of the Shift and Ctrl keys. To initiate a copy the user presses and holds down the mouse button, as well as the control key. The visual indication of this operation is the half toning of the drag image. Likewise, users can accomplish a link operation by pressing and holding down the mouse button, the Shift key, and the Ctrl key. The visual indication of this operation is a line that is drawn that connects the drag image with the object where the drag was initiated. Member functions that you use to override and restrict the drag operation are described in "Setting and Querying the Drag Operation" on page 453.

Users can request help during a direct manipulation operation by pressing F1, the help key. This displays help for the object a user is dragging over.

To cancel a direct manipulation operation, press the Esc (escape) key.

The User Interface Class Library provides four main classifications of objects to support direct manipulation:

- A drag item (IDMItem)
- A drag item provider (IDMItemProvider)
- A renderer (IDMSourceRenderer or IDMTargetRenderer)
- An event handler (IDMSourceHandler or IDMTargetHandler)

The collaboration of these objects allows the rendering from a source to a target location. *Rendering* is the process by which data is transferred from the source of a direct manipulation operation to a target. The following is an overview of the process:

1. A user initiates a drag request, which generates an event that is processed by the source handler.
2. The source handler uses the source window's attached drag item provider to request generation of source drag items.

   Drag items represent the objects that are the focus of the direct manipulation operation and provide access to the object's data. *Drag item providers* are designed to assist in the generation of drag items and bridge the gap between the drag items and the source and target handlers.

   When you use the IDMItemProviderFor template to instantiate the provider, the static function in the derived drag item class, generateSourceItems, is called. Alternatively, if the provider is instantiated from a derived drag item provider class, then the IDMItemProvider::provideSourceItems override is called.
3. Once the source drag items are generated, appropriate source renderers are selected.
4. When you enter a potential target window, an event is generated that is processed by the target handler.
5. The target handler uses the target window's attached drag item provider to request generation of target drag items.

   When you use the IDMItemProviderFor template to instantiate the provider, the target drag item constructor is called:

   ```
   IDMItem ( const Handle& item );
   ```

   Alternatively, if the provider is instantiated from a derived drag item provider class, then the IDMItemProvider::provideTargetItemFor override is called.
6. The target handler also uses the drag item provider to request additional verification support via the virtual function, IDMItemProvider::provideEnterSupport.
7. Once the target drag items are generated and verified, the appropriate target renderer is selected.
8. The drop is processed based upon the rendering mechanism and format, which is stored in the target renderer, and the data is subsequently transferred. The virtual function, IDMItem::targetDrop, is used to process the drop event.

Renderers encapsulate the various mechanisms and formats that are used for rendering and implement the logic that supports the mechanisms and formats (RMFs).

Generally, a *mechanism* defines the method of data transfer, whereas the *format* identifies the format of the data. The User Interface Class Library's implementation groups the mechanisms and formats under one mechanism.

See "Using Rendering Mechanisms and Formats" on page 426 for more information. The two different types of rendering are described below.

Target rendering transfers data from the source to the target. Because the source can package all the required transfer information when the drag operation begins, the target can complete the drop operation without further assistance from the source.

A good example of target rendering involves the use of the User Interface Class Library's process RMF. If the source and target are in the same process, the target directly accesses the source's data and subsequently renders the information as required.

On the other hand, *source rendering* occurs when the target requires additional information from the source in order to complete processing of the drop. The target issues the appropriate events to request the information from the source. A good example of source rendering involves the use of the User Interface Class Library's shared memory RMF. When the source and the target are in separate processes, the target can select this RMF that generates a shared memory buffer that, in turn, transfers the data between the source and target.

Using Default Direct Manipulation Support

User Interface Class Library provides default direct manipulation support for:

- Entry fields
- Multiple-line edit (MLE) fields
- Intra-process containers
- Tool bars (including menu bars and tool bar buttons)

Using Defaults for Entry Fields and MLEs

The default direct manipulation support for entry fields and MLEs is almost identical. When you create source items, selected text (or all of the text if none is selected) is stored within the item using the IDMItem::setContents function. Afterwards, the optimal rendering mechanisms and formats are determined using the length of the text characters plus any embedded characters. If the text length is fewer than or equal to 255 characters, and does not contain any embedded nulls, the text rendering format, IDM::rfText, is used. Otherwise, the shared rendering format, IDM::rfSharedMem is used. To further optimize performance, the process rendering format, IDM::rfProcess, is added to the prior selection to handle it when the source and target entry fields or

MLEs are located within the same process. If they are located within separate processes, one of the other rendering formats, is used.

The default rendering support for IDM::rfSharedMem automatically allocates the shared memory buffer and transfers the data, stored in the source item using IDMItem::setContents, from one process to another when its use is required. Therefore, the data is accessible to IDMItem::contents in the target item after the drop has occurred.

The default drag image style for the entry field and MLE support is IDM::allStacked.

The default implementation of the IDMItemProvider::provideEnterSupport function for the entry field and MLE items prevents a user from dropping text within the same window. The source and target window cannot be the same window.

When the entry field and MLE items differ, default target drop processing occurs. The entry field item retrieves the text using the IDMItem::contents function and sets the text into the entry field. The MLE item appends the text to the end of the MLE field.

Using Defaults for Containers

The default direct manipulation for containers supports moving or copying container objects within the same process. Also, all of the container views are supported. When you construct a source item, the container object is stored within the source item using the IDMItem::setObject function. Because a target can directly address a container object in the same process, you can use IDMItem::object at the target to access the container object after the drop has occurred. Therefore, the use of IDMItem::setContents and IDMItem::contents is supported if you extend the default support, but they are not used in the default implementation. Finally, the process rendering format, IDM::rfProcess, is set. This is the only RMF used for the default support.

If the user selects multiple container objects, a sequence collection is created. The source items are then stored based upon the following order:

1. The object under the mouse pointer is stored first.
2. The other objects are stored in the order that IContainerControl::ObjectCursor returns them.

The default drag image style for container support is IDM::allStacked.

The default implementation of the IDMItemProvider::provideEnterSupport function for the container item prevents the user from dropping an object on a target container object if drops have been disabled by the IContainerObject::disableDrop function.

Also, the IContainerControl::isMoveValid function is called to ensure that a move operation, the default, is valid.

Default target-drop processing handles both moving and copying. The default positioning of the dropped items is based upon the view of the target container. If multiple container objects are involved, IDMCnrItem::targetDrop is called once for each container item, and the items are processed in the reverse order in which they were added to the sequence collection.

When implementing container copy support, you must define an override for IContainerObject::objectCopy in the derived IContainerObject class. Also, you must define a copy constructor to be used by the override.

These are illustrated in the drag3 sample discussed in “Enabling Direct Manipulation for a Container” on page 431.

Enabling Default Support

You enable the default direct manipulation support for the container, entry field, and MLE by calling the desired IDMHandler static function. A pointer to the control object is supplied as the functional parameter to the static function, which performs all the required setup to enable the default support. Two samples, drag1 and drag3, are supplied to illustrate enabling default support.

The samples are described in “Enabling Direct Manipulation for an Entry Field or MLE” on page 430 and “Enabling Direct Manipulation for a Container” on page 431.

For a more general discussion of enablement, see “Enabling Drag and Drop” on page 439 which discusses how to enable a control that is not covered by the default support.

Containers and windows that support direct manipulation can be the source, target, or source and target of a drag operation. This is determined by the use of the static functions IDMHandler::enableDragFrom, IDMHandler::enableDropOn, or IDMHandler::enableDragDropFor, respectively. However, notice the differences in the support for the menu bar, tool bar, and tool bar buttons.

Using Defaults for Tool Bars

The default direct manipulation for tool bars supports the dropping of menu item objects from a menu bar within the same process to create a new tool bar button. Also, moving tool bar buttons within the same process is supported: you can move and arrange tool bar buttons within the same tool bar or you can move them from another tool bar. Deleting tool bar buttons is also supported, as you can drop the buttons on a Workplace Shell shredder object.

Default Support

When a source item is constructed from a menu item, the menu item resource identifier is stored within the source item using the IDMItem::setObject function, and the menu item text is stored using the IDMItem::setContents function. The process rendering format, IDM::rfProcess, is set as it is the only RMF used for the default support.

Afterwards, the IDMMenuItem constructor attempts to set the drag image based upon a stored image referenced by the resource identifier. The image can be one of the supplied IBM User Interface Class Library defaults or the user can define it. If one is unavailable, a default image is used. The default drag image style for menu bar support is IDM::allStacked. The default operation is IDMOperation::link, and the drag item type is IDM::menuItem.

Because the menu bar is only supported from a source perspective, the IDMItemProvider::provideEnterSupport and IDMItem::targetDrop functions are not implemented.

When a source item is constructed from a tool bar button, a pointer to the button window is stored within the source item. The process rendering format, IDM::rfProcess, is set, as well the shredder RMF, IDM::rmDiscard, and IDM::rfUnknown. Then, the IDMTBarButtonItem constructor sets the drag image based upon the button’s stored image. The image can be one of the supplied IBM User Interface Class Library defaults or the user can define it. If the image is unavailable, a default image is used. The default drag image style for tool bar button support is IDM::allStacked. The default operation is IDMOperation::move, and the drag item type is IDM::toolBarButton.

The default implementation of the IDMItemProvider::provideEnterSupport function for the tool bar button item prevents the user from dropping a button on itself. It also filters the drag item types to only allow drops for the following types: IDM::toolBarButton, IDM::menuItem, and IDM::bitmap. IDM::bitmap is included to allow system bitmaps to be dropped on a button. The special case of a system bitmap with a type of IDM::plainText, is also handled.

Default tool bar button drop processing handles both moving and linking. The default positioning of the dropped item is based upon the position of the object over the tool bar button when it was added to the tool bar. If the new button is dropped on the left half of a tool bar button, the button is moved before the button where the drop occurred. If the new button is dropped on the right half, or at the center of the tool bar button, the button is moved after the button where the drop occurred. This rule applies to every source of a drag operation, including tool bar buttons created from a menu bar, buttons within the same tool bar, and buttons from another tool bar. If the tool bar is vertical, a similar rule applies. If the new button is dropped on the lower half of a tool bar button, the new tool bar button is moved below the button where the drop occurred. If the new button is dropped on the upper half, or at the center of the tool bar button, the button is moved above the tool bar button where the drop occurred. Finally, if the item that was dropped was a system bitmap, the current tool bar button image is replaced using the system bitmap.

The tool bar only supports drop processing. The default positioning adds the tool bar button to the end of the tool bar and places it within its own group. If the source of the drag was a menu bar, a new button is created and added to the end of the tool bar. If the source of the drag was a tool bar button within the same tool bar, the button is moved to the end of the tool bar. When the source of the drag was a tool bar button in another tool bar, the button is removed from the source tool bar, and added to the end of the target tool bar.

Note: Tool bar support only works upon menu bars, tool bar buttons, and tool bars that are within the same process.

Understanding Drag Items

The first type of objects are the drag items, represented by objects of class IDMItem. *Drag items* encapsulate the logic that serves as the bridge between the context-insensitive handlers and renderers and the application-specific behavior of particular source and target windows. Thus, the drag items provide the application-specific semantics of the direct manipulation operation.

IDMItem is the base class that defines the general behavior of all direct manipulation items. The three derived classes provided by the User Interface Class Library provide specializations of the base class that represent the objects being dragged and dropped on specific controls.

The following classes are derived from IDMItem:

- IDMCnrItem
- IDMMLEItem
- IDMEFItem

Understanding Drag Item Provider

The IDMItemProvider class is an extension of the IWindow class that provides direct manipulation functions. Objects of the IDMItemProvider class allow generic controls, such as an entry field, to generate context-sensitive drag items. For example, a container that contains customer objects can generate a “customer” item; a bitmap can provide an item that can extract the picture from a .BMP file.

The IDMItemProvider class also provides functions that deal with target entry and exit, as well as help. With IDMItemProvider::provideEnterSupport you can verify objects, using different schemes, when the object is over a potential target.

Default Support

For example, if you want to restrict drops in an icon view container to the white space area of the container, you can use the provideEnterSupport function to determine if the pointing device is over either white space or an actual container object:

```
IBase::Boolean CnrProvider::provideEnterSupport
                          ( IDMTargetEnterEvent &event )
{
  /***********************************************************************/
  /* Allow default verification to occur                                 */
  /***********************************************************************/
  if (Inherited::provideEnterSupport( event ))
  {
    /*********************************************************************/
    /* Prevent the drop if we're over a container object (icon view)     */
    /*********************************************************************/
    if (event.object())
    {
      event.setDropIndicator( IDM::notOk );
      return(false);
    }
    return(true);
  }
  return(false);
}
```

Note: The return value is currently not used to determine if a drop is allowed. Only the drop indicator is used to set the drop disposition.

Refer to the drag2 sample for a more detailed example of creating and attaching a provider. See “Enabling a Control as a Drop Target” on page 439 for more information about providers.

Using Rendering Mechanisms and Formats

Rendering is the process by which data is transferred from the source of a direct manipulation operation to the target. If the source and target objects are within the same process, both objects have access to the same memory address space, and the target can readily access the source data to complete the transfer. If the source and target are in separate processes, the data transfer is facilitated using a shared memory buffer and an operation that involves the dispatching and processing of rendering messages.

Renderers transfer the representation of the object being manipulated from the source object to the target object. Direct manipulation renderers manage and maintain rendering mechanisms and formats (RMFs) whose characteristics are defined by the RMF pairs that represent the data transfer method. The rendering mechanisms and formats identify the set of protocols that your items support. These renderers are objects of classes IDMSourceRenderer and IDMTargetRenderer and are derived from IDMRenderer.

Default Support

When you create an IDMSourceHandler object, the User Interface Class Library creates a default IDMSourceRenderer. Table 6 displays the source RMF pairs and the corresponding drag item type. IDM::any represents any drag item type. Any object that you manipulate must have an explicit attribute that identifies the type of the item. These objects are often passed by Presentation Manager mechanisms that need to identify the attributes of an item.

Table 6. Default Source Renderer

Rendering Mechanism	Rendering Format	Item Type
IDM::rmLibrary	IDM::rfProcess	IDM::any
IDM::rmLibrary	IDM::rfText	IDM::any
IDM::rmLibrary	IDM::rfSharedMem	IDM::any
IDM::rmPrint	IDM::rfUnknown	IDM::any
IDM::rmDiscard	IDM::rfUnknown	IDM::any
IDM::rmFile	IDM::rfUnknown	IDM::any
IDM::rmObject	IDM::rfObject	IDM::any

When an IDMTargetHandler object is created, the User Interface Class Library creates a default IDMTargetRenderer. The default target renderer RMF pairs are shown in Table 7.

Table 7. Default Target Renderer

Rendering Mechanism	Rendering Format	Item Type
IDM::rmLibrary	IDM::rfProcess	IDM::any
IDM::rmLibrary	IDM::rfText	IDM::any
IDM::rmLibrary	IDM::rfSharedMem	IDM::any
IDM::rmFile	IDM::rfUnknown	IDM::any
IDM::rmObject	IDM::rfObject	IDM::any

The User Interface Class Library provides IDM::rmLibrary as the rendering mechanism used for efficient drag and drop operations. Table 8 displays other rendering messages defined as part of the default renderers.

Default Support

Table 8. Other Default Rendering Mechanisms

Rendering Mechanism	Used When...
IDM::rmPrint	An User Interface Class Library object is dropped on a printer
IDM::rmDiscard	An User Interface Class Library object is dropped on the shredder
IDM::rmFile	An OS/2 file is dragged from the source and dragged over or dropped on a target
IDM::rmObject	A Workplace Shell object is processed. Your application may be required to run under Workplace Shell process to use this rendering mechanism.

Several default rendering formats are defined to assist you, the application developer, in using the direct manipulation classes. Table 9 displays these default rendering formats..

Table 9. Default Rendering Formats

Format	Used When...
IDM::rfProcess	Determining if the source of the direct manipulation operation and the target are in the same process. This format must be constructed by calling the static member function IDMItem::rfForThisProcess.
IDM::rfText	Dragging text that has a length of 255 or fewer characters with no embedded null characters.
IDM::rfSharedMem	A shared memory buffer is required to transfer the data from the source to the target. This format should be used when transferring data between two separate processes and IDM::rfText cannot be used.
IDM::rfUnknown	The format is unknown.
IDM::rfObject	A Workplace Shell object is processed. Your application may be required to run under Workplace Shell process to use this rendering format.

Note: You can use IDM::any type to represent any drag item type.

The *native* renderer is the first rendering mechanism and format defined when you create the item. For example, in the declaration of the default source renderer, the native renderer supports the library rendering mechanism, the process rendering format, and any item type. In the declaration of the default target renderer, the native renderer supports the library rendering mechanism, the process rendering format, and any item type.

Table 10 displays the default User Interface Class Library RMF pairs that support target rendering. Table 11 displays the default User Interface Class Library RMF pairs that support source rendering.

Table 10. Target RMFs

Mechanism	**Format**
IDM::rmLibrary	IDM::rfProcess
IDM::rmLibrary	IDM::rfText
IDM::rmFile	IDM::rfUnknown
IDM::rmFile	IDM::rfText
IDM::rmObject	IDM::rfObject

Table 11. Source RMFs

Mechanism	**Format**
IDM::rmLibrary	IDM::rfSharedMem
IDM::rmPrint	IDM::rfUnknown
IDM::rmDiscard	IDM::rfUnknown

To create renderers for controls not supported by User Interface Class Library, you can create your own source or target renderer. To do this, derive from the IDMSourceRenderer or IDMTargetRenderer, create instances, and then add them to the handler using setDefaultTargetRenderer and setDefaultSourceRenderer.

Using Drag Item Types

Drag item types are useful in distinguishing drag items. Normally, the type is defined when the drag item object is constructed. IDMItem functions, such as IDMItem::setTypes and IDMItem::types, are defined to allow the setting and querying of the types, respectively.

The User Interface Class Library defines the following default types that you can use in your application:

IDM::any	Any type
IDM::binary	Generic binary item type
IDM::binaryData	Binary data item type
IDM::bitmap	Bitmap item type
IDM::container	Container item type

IDM::containerObject	Container object item type
IDM::file	File item type
IDM::icon	Icon item type
IDM::menuItem	Menu item drag item type
IDM::plainText	Plain text drag item type
IDM::text	Generic text drag item type
IDM::toolBarButton	Tool bar button drag item type
IDM::unknown	Unknown drag item type

You can define new types as required by your application if the above list does not have the types you need.

Enabling Direct Manipulation for an Entry Field or MLE

The following sample shows you how to enable direct manipulation for an entry field or an MLE control and how to use the static function, IDMHandler::enableDragDropFor. This static function creates the following:

- Source and target handlers
- Source and target default renderers
- An entry field item provider

In the following sample, the highlighted lines enable direct manipulation of text between two entry fields in the same process. Direct manipulation is enabled the same way for an MLE. The complete sample is located in the `\ibmcpp\samples\ioc\drag1` directory.

```
#include <iframe.hpp>
#include <ientryfd.hpp>
#include <idmefit.hpp>
#include <idmhndlr.hpp>

/*------------------------------------------------------------------------
| main                                                                    |
-------------------------------------------------------------------------*/
int main()
{

  /**********************************************************************/
  /* Create a generic frame window.                                     */
  /**********************************************************************/
  IFrameWindow
    frame( "ICLUI Direct Manipulation Sample 1" );

  /**********************************************************************/
  /* Create 2 entry fields for the client area.                         */
  /**********************************************************************/
21   IEntryField
22     client( 1000, &frame, &frame ),
23     ext   ( 1001, &frame, &frame );
24
25   /*********************************************************************/
26   /* Enable source and target direct manipulation support for both      */
27   /* entry fields.                                                      */
28   /*********************************************************************/
29   IDMHandler::enableDragDropFor( &client );
30   IDMHandler::enableDragDropFor( &ext );
31
32   /*********************************************************************/
33   /* Frame setup - Put both entry fields in the client area, with one   */
34   /* added as an extension.                                             */
35   /*********************************************************************/
36   frame
37     .setClient( &client )
38     .addExtension( &ext, IFrameWindow::belowClient, 0.5 )
39     .setFocus()
40     .show();
41
42   /*********************************************************************/
43   /* Run Direct Manipulation Sample 1                                   */
44   /*********************************************************************/
45   IApplication::current().run();
46
47 }
⋮
```

The preceding sample illustrates how you can enable direct manipulation if you only need default entry field support. If you substitute IMultiLineEdit for IEntryField in line 21, the above sample then demonstrates the default MLE support.

Enabling Direct Manipulation for a Container

This section shows you how to enable direct manipulation for a container and how to use the IDMHandler static functions enableDragFrom and enableDropOn.

In the following example, the dmsamp3.hpp file defines a container control object. The .CPP file creates the container and container objects and, in the highlighted lines, calls IDMHandler::enableDragFrom and IDMHandler::enableDropOn.

```
#include "dmsamp3.hpp"

/*----------------------------------------------------------------------
| main                                                                 |
-----------------------------------------------------------------------*/
int main()
{
   MySourceWin sourceWin (WND_SOURCE);
   MyTargetWin targetWin (WND_TARGET);
   IApplication::current().run();
}

:
```

Enabling Direct Manipulation

```
.
MySourceWin :: MySourceWin ( unsigned long windowId ) :
               MyWindow ( windowId )
{
  ITitle title (this, "C Set ++ Direct Manipulation - Source Container" );

  /***********************************************************************/
  /* Enable the source for dragging from (only).                         */
  /***********************************************************************/
  IDMHandler::enableDragFrom( cnrCtl );
};

/*-------------------------------------------------------------------------
| MyTargetWin::MyTargetWin                                    |
|                                                                         |
| Constructor.                                                            |
-------------------------------------------------------------------------*/
MyTargetWin :: MyTargetWin ( unsigned long windowId ) :
               MyWindow ( windowId )
{
  ITitle title (this, "C Set ++ Direct Manipulation - Target Container" );

  /***********************************************************************/
  /* Enable the target for dropping on (only).                           */
  /***********************************************************************/
  IDMHandler::enableDropOn( cnrCtl );
}
⋮
/*------------------------------------------------------------------------
| Customer::Customer                                                     |
|                                                                        |
| Copy constructor.                                                      |
------------------------------------------------------------------------*/
Customer :: Customer ( const Customer &cnrobj )  :
            IContainerObject ( (const IContainerObject &)cnrobj ),
            strName ( cnrobj.name() ),
            strAddress ( cnrobj.address() ),
            strPhone ( cnrobj.phone() ),
            myWin ( cnrobj.myWin )
{
}
⋮
/*------------------------------------------------------------------------
| Customer::objectCopy                                                   |
|                                                                        |
| Make a copy of the Customer object.  Called by                         |
| IContainerObject::copyObjectTo().                                      |
------------------------------------------------------------------------*/
IContainerObject* Customer :: objectCopy()
{
  /***********************************************************************/
  /* Use Customer copy constructor to make a copy of the object.         */
  /***********************************************************************/
  Customer *copy = new Customer(*this);
  return((IContainerObject *)copy);
}
⋮
```

Lines 81 through 84 create a source window.

Line 89 enables the window as a drag source.

Lines 97 through 100 create a target window.

Line 105 enables the window as a drop target.

Lines 113 through 120 implement the copy constructor that is used by the Customer::objectCopy function.

Lines 147 through 154 implement the IContainerObject::objectCopy override function, Customer::objectCopy.

The preceding sample illustrates how you can enable direct manipulation if you only need default container support. The complete sample is located in the `\ibmcpp\samples\ioc\drag3` directory.

The previous container example only illustrates intraprocess (source and target containers are in the same process) container support. The following sample shows inter process (source and target containers are in separate processes) container support. The module dmsamp4.cpp contains the key logic for the drag4 sample.

Note: You must start two copies of the drag4 sample to view the interprocess support.

```
⋮
  3 /*---------------------------------------------------------------------------
  4 | main                                                                      |
  5 ---------------------------------------------------------------------------*/
  6 int main( int argc, char* argvﬄ" )
  7 {
  8   /*************************************************************************/
  9   /* Permit debugging during the drag                                      */
 10   /*************************************************************************/
 11   IDM::debugSupport = true;
 12
 13   /*************************************************************************/
 14   /* Create window                                                         */
 15   /*************************************************************************/
 16   DMSample4Window
 17     frame( argvﬄ1" );
 18
 19   /*************************************************************************/
 20   /* Show it                                                               */
 21   /*************************************************************************/
 22   frame.showModally();
 23 }
 24
 25 /*---------------------------------------------------------------------------
 26 | CustomerItem::CustomerItem                                                |
 27 |                                                                           |
| Constructor.                                                            |
-------------------------------------------------------------------------*/
CustomerItem :: CustomerItem ( const IDMItem::Handle& item ) :
                IDMCnrItem ( item )
{
  IString
    rmf1 = IDMItem::rmfFrom( IDM::rmLibrary, IDM::rfSharedMem ),
    rmf2 = IDMItem::rmfFrom( IDM::rmDiscard, IDM::rfUnknown );

  /***********************************************************************/
  /* Get pointer to the associated Customer container object             */
  /***********************************************************************/
  Customer *pCustomer = (Customer *)object();

  /***********************************************************************/
  /* Build and set contents.  We can only do this on the source          */
  /* side.  Note that since we call this constructor on both source      */
  /* and target sides, we must distinguish them.  That is done           */
  /* here by checking the "object" pointer.  If this constructor was     */
  /* called from within our generateSourceItems, then this value         */
  /* will be nonzero.  If called from with the template provider's       */
  /* provideTargetItemFor, then it will be 0.                            */
  /***********************************************************************/
  if (pCustomer)
  {
    IString
      contents,
      delim = '\x01';

    contents += pCustomer->iconText() + delim;
    contents += pCustomer->name() + delim;
    contents += pCustomer->address() + delim;
    contents += pCustomer->phone() + delim;
    contents += pCustomer->iconId();

    setContents( contents );

    /*********************************************************************/
    /* Add RMFs supported by this class (IDMCnrItem will have            */
    /* already specified the other RMFs we use).                         */
    /*********************************************************************/
    addRMF( rmf1 );
    addRMF( rmf2 );
  }
  else
  {
    /*********************************************************************/
    /* On target side, add in <rmLibrary,rfSharedMem> if source concurs  */
    /* (and it's not already in there).                                  */
    /*********************************************************************/
    if ((item->supportsRMF( rmf1 ))  &&
        !(supportsRMF( rmf1 )))
    {
      addRMF( rmf1 );
    }
  }
}

```

```
/*-------------------------------------------------------------------------
| CustomerItem::generateSourceItems                                       |
|                                                                         |
| Called to give CustomerItem opportunity to attach new CustomerItem's to |
| the argument IDMSourceOperation object.                                 |
-------------------------------------------------------------------------*/
IBase::Boolean CustomerItem :: generateSourceItems ( IDMSourceOperation* pSrcOp )
{
  /***********************************************************************/
  /* Get generic container items.  Note that we call the inherited       */
  /* function since it already has logic to deal with multiselection,    */
  /* etc.                                                                */
  /***********************************************************************/
  Boolean result = Inherited::generateSourceItems( pSrcOp );

  /***********************************************************************/
  /* Now, replace each IDMCnrItem with a CustomerItem.                   */
  /***********************************************************************/
  for (unsigned i = 1; i <= pSrcOp->numberOfItems(); i++)
  {
    pSrcOp->replaceItem( i, new CustomerItem( pSrcOp->item( i ) ) );
  }

  /***********************************************************************/
  /* Set stack3AndFade as the default image style and set the stacking   */
  /* percentage that is used to set the stacking offset as a percentage  */
  /* of the image size.                                                  */
  /***********************************************************************/
  pSrcOp->setImageStyle( IDM::stack3AndFade );
  pSrcOp->setStackingPercentage( IPair( 25, 25 ) );
  return( result );
}

/*-------------------------------------------------------------------------
| CustomerItem::supportedOperationsFor                                    |
|                                                                         |
| Called when a CustomerItem is dropped on a target container.            |
-------------------------------------------------------------------------*/
unsigned long CustomerItem ::
              supportedOperationsFor ( const IString& rmf ) const
{
  if (rmf == IDMItem::rmfFrom( IDM::rmLibrary, IDM::rfSharedMem ))
  {
    /*********************************************************************/
    /* If using <rmLibrary,rfSharedMem> then only copy is supported      */
    /*********************************************************************/
    return( IDMItem::copyable & supportedOperations() );
  }

  /***********************************************************************/
  /* Otherwise, whatever base class supports                             */
  /***********************************************************************/
  return( Inherited::supportedOperationsFor( rmf ) );
}

/*-------------------------------------------------------------------------
| CustomerItem::sourceDiscard                                             |
|                                                                         |
| Called when a CustomerItem is dropped on a Workplace Shell shredder.    |
-------------------------------------------------------------------------*/
IBase::Boolean CustomerItem :: sourceDiscard ( IDMSourceDiscardEvent& event )
{
  /***********************************************************************/
  /* Remove the object from the container.                               */
  /***********************************************************************/
  IContainerControl
   *pCnr = (IContainerControl *)(event.sourceOperation()->sourceWindow());
  IContainerObject
   *pCnrObj = (IContainerObject *)(object());

  pCnr->removeObject( pCnrObj );
  return( true );
}

/*-------------------------------------------------------------------------
| CustomerItem::targetDrop                                                |
|                                                                         |
| Called when a CustomerItem is dropped on a target container.            |
-------------------------------------------------------------------------*/
IBase::Boolean CustomerItem :: targetDrop ( IDMTargetDropEvent& event )
{
  Boolean result = true;

  /***********************************************************************/
  /* Check if using ICLUI shared memory rendering format                 */
  /***********************************************************************/
  IString myRMF = IDMItem::rmfFrom( IDM::rmLibrary, IDM::rfSharedMem );
  if (selectedRMF() == myRMF)
  {
    /*********************************************************************/
    /* Yes, construct new Customer object from passed data.              */
    /*********************************************************************/
    IString
      contents = this->contents(),
      delim    = '\x01',
      text     = contents.subString( 1, contents.indexOf( delim ) - 1 );

    contents   = contents.subString( contents.indexOf( delim ) + 1 );
    IString
      name     = contents.subString( 1, contents.indexOf( delim ) - 1 );

    contents   = contents.subString( contents.indexOf( delim ) + 1 );
    IString
      addr     = contents.subString( 1, contents.indexOf( delim ) - 1 );

    contents   = contents.subString( contents.indexOf( delim ) + 1 );
    IString
      phone    = contents.subString( 1, contents.indexOf( delim ) - 1 ),
      iconId   = contents.subString( contents.indexOf( delim ) + 1 );

    IContainerControl *pCnr = event.container();
    Customer *pNewCustomer = new Customer( text,
                                           iconId.asUnsigned(),
                                           name,
                                           addr,
                                           phone,
                                           (MyWindow *)(pCnr->parent()) );

    /*********************************************************************/
    /* Insert the new Customer object into the container.                */
    /*********************************************************************/
    pCnr->addObject( pNewCustomer );

    /*********************************************************************/
    /* Create an IDMItem::Handle                                         */
    /*                                                                   */
    /*  We must break this into 2 statements due to a bug in the         */
    /*  IRefCounted class.  If we use an initializer to create           */
    /*  the handle, this sample will eventually trap due to the          */
    /*  inability of the initializer to properly increment the           */
    /*  drag item object use count:                                      */
    /*  IDMItem::Handle thisHandle = this; //initializer form            */
    /*                                                                   */
    /*  When we break the create into 2 statements, it takes the         */
    /*  form of an assignment which does not have the problem.           */
    /*********************************************************************/
    IDMItem::Handle thisHandle;
    thisHandle = this;

    /*********************************************************************/
    /* Position the object within the container.                         */
    /*********************************************************************/
    IPoint pos = targetOperation()->dropPosition( thisHandle, event );
    pCnr->moveObjectTo( pNewCustomer,
                        0,
                        pCnr,
                        0,
                        pos );
  }
  else
  {
    /*********************************************************************/
    /* Some other RMF, base class must support it.                       */
    /*********************************************************************/
    result = Inherited::targetDrop( event );
  }

  return( result );
}

/*-------------------------------------------------------------------------
| DMSample4Window::DMSample4Window                                        |
|                                                                         |
| Constructor.                                                            |
-------------------------------------------------------------------------*/
DMSample4Window :: DMSample4Window ( const char* aTitle ) :
                   MyWindow( 0 ),
                   title( this )
{
  /***********************************************************************/
  /* Set the title.                                                      */
  /***********************************************************************/
  if (aTitle)
    title.setTitleText( aTitle );
```

```
260   else
261     title.setTitleText( "Direct Manipulation Sample 4" );
262
263   /*********************************************************************/
264   /* Tailor the container.                                             */
265   /*********************************************************************/
266   this->cnrCtl->hideTitle();
267   this->cnrCtl->showIconView();
268   this->cnrCtl->arrangeIconView();
269   this->cnrCtl->setExtendedSelection();
270
271   /*********************************************************************/
272   /* Set the item provider.                                            */
273   /*********************************************************************/
274   this->cnrCtl->setItemProvider( &this->provider );
275
276   /*********************************************************************/
277   /* Enable drag/drop.                                                 */
278   /*********************************************************************/
279   IString sTitle = aTitle;
280   if (sTitle.includes( "source only" ))
281     IDMHandler::enableDragFrom( this->cnrCtl );
282   else
283   {
284     if (sTitle.includes( "target only" ))
285       IDMHandler::enableDropOn( this->cnrCtl );
286     else
287       IDMHandler::enableDragDropFor( this->cnrCtl );
288   }
289
290   /*********************************************************************/
291   /* Resize the container.                                             */
292   /*********************************************************************/
293   this->sizeTo( ISize( 250, 275 ) );
294 }
⋮
```

The User Interface Class Library's shared memory rendering format provides the interprocess support. The shared memory format uses a shared memory buffer to transfer the container object data that is stored in the source item (using IDMItem::setContents) to the target item where the data can be retrieved (using IDMItem::contents). Remember that the source item and target items are in separate processes.

Two of the functions used in the previous example, CustomerItem::sourceDiscard on lines 146 through 158 and CustomerItem::supportedOperationsFor on lines 124 through 139, need more explanation. The sourceDiscard function demonstrates container object removal after the user drops the object on the Workplace Shell shredder. The supportedOperationsFor function lets you determine which operation or operations a drag item supports based upon the selected rendering mechanism and format. For example, you could make the item IDMItem::copyable, as shown in the preceding example, if the selected RMF is the User Interface Class Library shared memory RMF. For other RMFs, you could let the drag item default to IDMItem::moveable.

The CustomerItem constructor shown in lines 30 through 84 is generally used to construct target items because it is automatically called for target item construction when using the IDMItemProviderFor template. However, this sample shows how to use it to construct source items as well. The CustomerItem constructor uses the IDMItem::object function to determine if a source or a target item is being constructed. Line 106 in the CustomerItem::generateSourceItems function is the key, as it calls the constructor to create the source item.

Samples are provided with the User Interface Class Library product. Complete listings of the samples used in this chapter are included in the `\ibmcpp\samples\ioc` directory unless otherwise noted. This sample is found in `\ibmcpp\samples\ioc\drag4` directory.

Enabling Drag and Drop

OS/2 Presentation Manager (PM) implements direct manipulation using a set of window events specific to that task. If your User Interface Class Library applications support drag and drop, you must attach handlers to your windows to process those events. The direct manipulation handlers implement each of the handler virtual functions to compose a functional framework you can use as provided or you can extend. You do not derive from these classes but you derive from the other classes in the direct manipulation framework.

IDMSourceHandler and IDMTargetHandler are derived from IDMHandler. They handle the Presentation Manager direct manipulation window messages. Objects from these classes pick up the WM_* and DM_* messages for the source and target objects and translate them into virtual function calls to the handler.

In addition to translating messages to virtual function calls, these handlers also manage renderers.

The following sections discuss how to add direct manipulation to your applications.

Enabling a Control as a Drop Target

To enable other controls as drop targets, you must specifically create the drag items and item providers that the User Interface Class Library generates automatically for entry field, MLE, and container controls. Do the following:

1. Derive a class from the base class IDMItem and override the targetDrop member function and the following IDMItem constructor:

```
    IDMItem ( const Handle&       item );
```

2. Write a drag item provider class for the customized item class using the IDMItemProviderFor template class, overriding provider functions when necessary:

   provideEnterSupport
   Override to provide drag item and target verification that is not supplied by default and provide target emphasis.

   provideLeaveSupport
   Override to provide additional cleanup not supplied by default and remove target emphasis.

   provideHelpFor
   Override to provide help support.

3. Use the default target handler and renderer for the customized object.

4. Use the static function, IDMHandler::enableDropOn to enable the control as a drag target as shown in line 35 of the .CPP sample file that follows.

5. Instantiate and set the drag item provider for for the control as shown in lines 40, 41, and 46 of the .CPP sample file below.

Note: When implementing a specialized drop target for a container, entry field, or MLE control, you should derive from that control specific class instead of IDMItem.

The following example adds drop support to a bitmap control. The header file dmsamp2.hpp defines two classes, ABitmapItem and ABitmapProvider, and overrides the IDMItem::targetDrop and IDMItem::provideEnterSupport member functions in both classes, respectively.

```
#include <idmprov.hpp>
#include <idmitem.hpp>
#include <idmevent.hpp>

class ABitmapItem : public IDMItem {
/*************************************************************************
* Objects of this class provide "bitmap control" drop behavior when a      *
* source bitmap file is dropped on a bitmap control properly configured    *
* with a target handler and an ABitmapProvider.                            *
*************************************************************************/
public:
/*------------------------------- Constructor ------------------------------
| Objects of this class are constructed from a generic item handle.        |
------------------------------------------------------------------------*/
  ABitmapItem ( const IDMItem::Handle &item );

/*------------------------------ Drop Behavior -----------------------------
| targetDrop - Take the dropped file, create a PM bitmap object,           |
|              and set it into the target window.                          |
------------------------------------------------------------------------*/
21 virtual Boolean
22   targetDrop ( IDMTargetDropEvent & );
23 };
24
25 class ABitmapProvider : public IDMItemProviderFor< ABitmapItem > {
26 /**************************************************************************
27 * Objects of this class are attached to bitmap controls in order to have    *
28 * ABitmapItem objects created when a bitmap file is dropped on such a       *
29 * control.                                                                  *
30 **************************************************************************/
31 public:
32 /*------------------------------ Target Support ---------------------------
33 |provideEnterSupport - Verify that we're dealing with a bitmap object before|
34 |                      allowing a drop, as well as draw the target emphasis.|
35 |provideLeaveSupport - Remove the target emphasis.                          |
36 |drawEmphasis        - Draw or remove the target emphasis.                  |
37 -------------------------------------------------------------------------*/
38 virtual Boolean
39   provideEnterSupport ( IDMTargetEnterEvent& event ),
40   provideLeaveSupport ( IDMTargetLeaveEvent& event );
41
42 virtual ABitmapProvider
43  &drawEmphasis        ( IBitmapControl*      bitmapControl,
44                         IDMTargetEvent&      event,
45                         Boolean              draw = true );
46 private:
47 static Boolean
48   bAlreadyDrawn;
49
50 };
```

Lines 5 through 23 declare IDMItem as the base class for objects of a specialized class named ABitmapItem. Objects of this class provide bitmap control drop behavior when a source bitmap file is dropped on a bitmap control that is properly configured with a target handler and an ABitmapProvider.

Lines 25 through 50 define a drag item provider for a bitmap control and override IDMItemProvider::provideEnterSupport so that it verifies that the dragged object is a bitmap.

The .CPP file adds the drag item provider and the target handler, and it uses the default target renderer.

The following code continues this same example:

```
⋮
 17 int main()
 18 {
 19   /*******************************************************************/
 20   /* Create a generic frame window.                                   */
 21   /*******************************************************************/
 22   IFrameWindow
 23     frame ( "C Set ++ Direct Manipulation - Sample 2" );
 24
```

```
  /***********************************************************************/
  /* Create an empty bitmap control.                                     */
  /***********************************************************************/
  IBitmapControl
    bmpControl ( 0, &frame, &frame );

  /***********************************************************************/
  /* Create a target handler for the bitmap control and use default      */
  /* C Set++ UI renderers.                                               */
  /***********************************************************************/
  IDMHandler::enableDropOn( &bmpControl );

  /***********************************************************************/
  /* Create bitmap drag item provider.                                   */
  /***********************************************************************/
  ABitmapProvider
    itemProvider;

  /***********************************************************************/
  /* Attach provider to the bitmap control.                              */
  /***********************************************************************/
  bmpControl.setItemProvider( &itemProvider );

  /***********************************************************************/
  /* Set the bitmap's control as the frame's client                      */
  /* and display the frame.                                              */
  /***********************************************************************/
  bmpControl.setText( "Drop .bmp files here." );
  bmpControl.setFocus();

  frame.sizeTo( ISize( 400, 350 ) );
  frame.setClient( &bmpControl )
       .showModally();
}

/*--------------------------------------------------------------------------
| ABitmapItem::ABitmapItem                                                 |
|                                                                          |
| Constructor.                                                             |
--------------------------------------------------------------------------*/
ABitmapItem :: ABitmapItem ( const IDMItem::Handle& item )
  : IDMItem( item )
{
}

/*--------------------------------------------------------------------------
| ABitmapItem::targetDrop                                                  |
|                                                                          |
| Take the dropped file, create a PM bitmap object,                        |
| and set it into the target window.                                       |
--------------------------------------------------------------------------*/
IBase::Boolean ABitmapItem :: targetDrop ( IDMTargetDropEvent& event )
{
  /*********************************************************************/
  /* Get pointer to the target bitmap control.                         */
  /*********************************************************************/
  IBitmapControl
   *bmpControl = (IBitmapControl *)targetOperation()->targetWindow();

  /***********************************************************************/
  /* Turn off target emphasis.                                           */
  /***********************************************************************/
  ABitmapProvider
   *provider = (ABitmapProvider *)bmpControl->itemProvider();
  provider->drawEmphasis( bmpControl, event, false );

  /***********************************************************************/
  /* Construct dropped .BMP file name from this drag item and attempt to */
  /* load the bitmap from a system file                                  */
  /***********************************************************************/
  IString
    fname = containerName() + sourceName();
  IGBitmap
    bitMap( fname );

  /***********************************************************************/
  /* If bitmap was successfully loaded, then set it.  Note that the old  */
  /* one will be automatically deleted.                                  */
  /***********************************************************************/
  if (bitMap.handle())
  {
    bmpControl->setBitmap( bitMap.handle() );

    /********************************************************************/
    /* Indicate name of dropped file.                                   */
    /********************************************************************/
    bmpControl->setText( fname );
  }
  else
  {
    bmpControl->setText( "Couldn't create bitmap!" );
  }

  return( true );
}

/*-------------------------------------------------------------------------
| ABitmapProvider::provideEnterSupport                                    |
|                                                                         |
| Verify that we're dealing with a bitmap object before                   |
| allowing a drop, as well as draw the target emphasis.                   |
-------------------------------------------------------------------------*/
IBase::Boolean ABitmapProvider :: provideEnterSupport ( IDMTargetEnterEvent& event )
{
  /***********************************************************************/
  /* Get handle to the drag target operation                             */
  /***********************************************************************/
  IDMTargetOperation::Handle targetOp = IDMTargetOperation::targetOperation();

  /***********************************************************************/
  /* Get pointer to the target bitmap control.                           */
  /***********************************************************************/
  IBitmapControl
   *bmpControl = (IBitmapControl *)event.window();

  /***********************************************************************/
  /* Draw the target emphasis.                                           */
  /***********************************************************************/
  drawEmphasis( bmpControl, event );

  /***********************************************************************/
  /* Filter the types of items that we allow to be dropped.              */
  /***********************************************************************/
  IDMItem::Handle pTgtDIH = targetOp->item(1);
  IString strTypes = pTgtDIH->types();

  /***********************************************************************/
  /* If type is either "Bitmap" or "Plain Text" (used by WPS), we can    */
  /* display the drag item.  If type is "Plain Text", then filter based  */
  /* upon the ".bmp" extension.                                          */
  /***********************************************************************/
  if (strTypes.indexOf( IDM::bitmap ))
  {
    return( true );
  }
  else
  {
    if ((strTypes.includes( IDM::plainText ))  &&
        (pTgtDIH->sourceName().lowerCase().includes( ".bmp" )))
    {
      return( true );
    }
  }

  /***********************************************************************/
  /* Type is not recognized - set the drop indicator to prevent a drop!  */
  /***********************************************************************/
  event.setDropIndicator( IDM::neverOk );
  return( false );
}

/*-------------------------------------------------------------------------
| ABitmapProvider::provideLeaveSupport                                    |
|                                                                         |
| Remove the target emphasis.                                             |
-------------------------------------------------------------------------*/
IBase::Boolean ABitmapProvider :: provideLeaveSupport (IDMTargetLeaveEvent& event)
{
  /***********************************************************************/
  /* Get pointer to the target bitmap control.                           */
  /***********************************************************************/
  IBitmapControl
   *bmpControl = (IBitmapControl *)event.window();

  /***********************************************************************/
  /* Remove the target emphasis.                                         */
  /***********************************************************************/
  drawEmphasis( bmpControl, event, false );
  return(true);
}

/*-------------------------------------------------------------------------
| ABitmapProvider::drawEmphasis                                           |
|                                                                         |
| Draw/remove the target emphasis                                         |
-------------------------------------------------------------------------*/
ABitmapProvider& ABitmapProvider::drawEmphasis ( IBitmapControl* bmpControl,
                                                 IDMTargetEvent& event,
                                                 Boolean bDraw )
{
  /***********************************************************************/
  /* Return if the request is to draw the emphasis, and its already      */
  /* drawn.                                                              */
  /***********************************************************************/
  if (bDraw && bAlreadyDrawn)
    return( *this );

  if (bDraw)
    bAlreadyDrawn = true;
  else
    bAlreadyDrawn = false;

  /***********************************************************************/
  /* Create the graphic context and set the mix mode to cause the pen    */
  /* color to be the inverse of the drawing surface.  Also, set the draw */
  /* operation, so that only the frame of the rectangle is drawn.        */
  /***********************************************************************/
  IGraphicContext
    graphicContext( event.presSpace() );

  graphicContext.setMixMode( IGraphicBundle::invert )
                .setDrawOperation( IGraphicBundle::frame );

  /***********************************************************************/
  /* Define the points for the emphasis rectangle and adjust their       */
  /* position so the rectangle will fit within the control window.       */
  /***********************************************************************/
  IPoint
    origin  ( bmpControl->rect().left(),  bmpControl->rect().bottom() ),
    topRight( bmpControl->rect().width(), bmpControl->rect().height() );

  origin -= 2;
  topRight -= 4;

  /***********************************************************************/
  /* Create an IRectangle object based upon the points defined and use   */
  /* it to construct a 2-Dimensional rectangle object: IGRectangle.      */
  /* Draw the emphasis rectangle using the IGRectangle object.           */
  /***********************************************************************/
  IGRectangle
    graphicRectangle( IRectangle( origin, topRight ) );

  graphicRectangle.drawOn( graphicContext );

  /***********************************************************************/
  /* Release presentation space and return.                              */
  /***********************************************************************/
  event.releasePresSpace();
  return( *this );
}
⋮
```

Enabling Direct Manipulation

First, the .CPP file creates an empty bitmap control object, bmpControl, and then creates and attaches the handler, provider, and renderer.

Lines 28 and 29 create the bitmap control object.

Line 35 constructs a target handler, which creates a default target renderer.

Lines 40 and 41 construct a drag item provider, itemProvider.

Line 46 attaches the drag item provider to bmpControl window.

The rest of the .CPP file defines the overridden member functions, IDMItem::targetDrop and IDMItemProvider::provideEnterSupport, for the classes declared in the .HPP file.

Lines 76 through 119 define AbitmapItem::targetDrop. This member function gets the dropped file, creates the bitmap, and displays the bitmap in the target window.

Lines 127 through 174 define ABitmapProvider::provideEnterSupport. This member function verifies that the object over the target is a bitmap.

Note: This data type verification is in addition to the RMF checking that is done by the User Interface Class Library default target renderer. IDM::plainText is also verified. This data type is used by the Workplace Shell for its background bitmaps.

If it is not a bitmap, the drop is not allowed.

Also, on line 143, provideEnterSupport requests the drawing of the target emphasis, which is a rectangle that is drawn just inside the outer edge of the target window. This member function is called when a target enter event (IDMTargetEnterEvent) occurs on a target window.

Lines 181 through 194 define ABitmapProvider::provideLeaveSupport. This member function requests removal of the target emphasis since the object is no longer over the target window. This member function is called when a target leave event (IDMTargetLeaveEvent) occurs on a target window.

Lines 201 through 254 define drawEmphasis. This member function has been added to the derived item provider class, ABitmapProvider, to draw and remove the target emphasis from the target window.

Note: When creating a graphic context, as shown on lines 222 and 223, you must use the presentation space obtained by calling IDMTargetEvent::presSpace. You cannot use the presentation space returned by IWindow::presSpace.

The rest of this function illustrates how to draw and remove the target emphasis using the graphic support classes.

A complete listing of this sample is included in the `\ibmcpp\samples\ioc\drag2` directory.

Enabling a Control as a Drag Source

To enable other controls as drag sources, you must specifically create the drag items and item providers that the User Interface Class Library generates automatically for entry field, MLE, and container controls. You should do the following:

1. Derive a class from the base class IDMItem, implement the generateSourceItems static member function, and override the following IDMItem constructors:

   ```
   IDMItem ( IDMSourceOperation* sourceOperation,
             const IString&      types,
             const unsigned long supportedOperations = unknown,
             const unsigned long attributes = none);

   IDMItem ( const Handle&       item );
   ```

   The second constructor, the target item constructor, is required by the IDMItemProviderFor template whether it is used or not.

2. Write a drag item provider class for the customized item class using the IDMItemProviderFor template class.

3. Use the default source handler and renderer for the customized object.

4. Use the static function, IDMHandler::enableDragFrom to enable the control as a drag source as shown in line 17 of the sample that follows.

5. Instantiate and set the drag item provider for the control as shown in lines 21 and 22 of the sample that follows.

Note: When implementing a specialized drag source for a container, entry field, or MLE control, you should derive from that control specific class instead of IDMItem.

The following example enables the user to drag objects from a static text control. This example is not included in the `\ibmcpp\samples\ioc` directory.

The header file defines two classes, STextItem and MyWindow, and implements the IDMItem::generateSourceItems static member function.

```
⋮
 9 #include "static.h"
10
11 class STextItem : public IDMItem {
12 public:
13
```

```
  STextItem ( IDMSourceOperation    *pSrcOp );
  STextItem ( const IDMItem::Handle &item );

static Boolean
  generateSourceItems ( IDMSourceOperation *pSrcOp );
};

class MyWindow : public IFrameWindow {
public:

  MyWindow();
  ~MyWindow();

private:
  ITitle title;
  ISetCanvas canvas;
  IStaticText staticText;
};
```

The .CPP file adds the drag item provider and the source handler, and it uses the default source renderer.

```
⋮
int  main()
{
  MyWindow myWin;
  IApplication::current().run();
}

MyWindow :: MyWindow( )  :
            IFrameWindow( ID_MYWINDOW ),
            title( this, "Static Control" ),
            canvas( ID_CANVAS, this, this ),
            staticText( ID_STEXT, &canvas, &canvas )
{
   setClient (&canvas);   //Set the canvas as the frame client.

   IDMHandler::enableDragFrom (&staticText); //Enable the static text for dragging from

   // Use the IDMItemProviderFor template class to create a template
   // for the static text item, and set it into the window.
   IDMItemProvider *pSTProvider = new IDMItemProviderFor< STextItem >;
   staticText.setItemProvider (pSTProvider);

   staticText.setText ("Static Text");  //Put text into the static text control.
   setFocus ();                         // Set the keyboard focus and show it.
   show ();
}

MyWindow :: ~MyWindow() {};
STextItem :: STextItem (IDMSourceOperation *pSrcOp)  :
            IDMItem (pSrcOp,
                      IDM::text,
                      (IDMItem::moveable | IDMItem::copyable),
                      none)
{
   IStaticText *pSText = (IStaticText *)pSrcOp->sourceWindow(); //Get a pointer to the static text
                                  // control from the source operation
38    setContents(pSText->text());   // Store the static text within the item
39    setRMFs(rmfFrom(IDM::rmLibrary, IDM::rfText));//Use the default RMF for text
40 }
41
42 STextItem :: STextItem (const IDMItem::Handle &item) :
43                          IDMItem ( item ) {};
44
45 IBase::Boolean STextItem :: generateSourceItems(IDMSourceOperation *pSrcOp)
46 {
47   STextItem *pSTItem = new STextItem (pSrcOp);   //Create the static text drag item
48   pSrcOp->addItem (pSTItem);                     //and add it to the source operation.
49
50   return(true);
51 }
```

First, the .CPP file creates a canvas and a static text control object, staticText, and then creates and attaches the handler, provider, and renderer. The renderer is automatically created and attached by the handler.

Line 13 creates the static text control object.

Line 17 constructs a target handler, which creates a default target renderer.

Lines 21 and 22 construct a drag item provider, *pSTProvider*, and attach the drag item provider to the static text window.

The rest of the .CPP file implements the static member function, STextItem::generateSourceItems, and the source and target item constructors.

Lines 30 through 40 define the source item constructor.

Lines 42 and 43 define the target item constructor, which is required by the IDMItemProviderFor template whether it is used or not. Note that this constructor can also be used to construct source items but requires additional work as shown in the drag4 sample.

Lines 45 through 51 define generateSourceItems. This member function is used to generate the source item.

Note: This sample is not included in the `\ibmcpp\samples\` directory.

Enabling a Control as a Drag Source and a Drop Object

To enable other controls as a drag source and a drop target, you must specifically create the drag items and item providers that the User Interface Class Library generates automatically for entry field, MLE, and container controls.

To enable a control as both a drag source and a drop object, follow the requirements in the following sections:

- "Enabling a Control as a Drop Target" on page 439
- "Enabling a Control as a Drag Source" on page 447

Note: Enable your control as a drag source and a drop target using the static function, IDMHandler::enableDragDropFor.

Enabling a Control to Support a Workplace Shell Shredder Object

When a user drop an object on a Workplace Shell shredder, two events are issued: a discard event and an end event. Do the following to enable your control to support a Workplace Shell shredder:

1. Add the shredder RMF, IDM::discard, IDM::rfUnknown to the control's derived drag item. This normally occurs within the derived class's source item constructor.
2. Override the IDMItem::sourceDiscard member function in the control's derived drag item to remove the object from the source window.
3. Override the IDMItem::sourceEnd member function in the control's derived drag item, if needed, to perform any additional clean-up.

The Workplace Shell shredder will set the drag operation to a value of IDMOperation::move. To prevent confusion with an actual move operation, you can set a flag in the control's derived drag item, from within the sourceDiscard override, to clearly identify the discard operation.

Note: If a user drops multiple items on a shredder, the Workplace Shell issues a discard and an end event for each item.

Enabling a Control to Support a Workplace Shell Printer Object

When an object is dropped on a Workplace Shell printer object, it issues two events: a print event, and an end event. Do the following to enable any control to support a Workplace Shell printer object:

1. Add the print RMF, IDM::print, IDM::rfUnknown, to the control's derived drag item. This normally occurs within the derived class's source item constructor.
2. Override the IDMItem::sourcePrint member function in the control's derived drag item to print the object.
3. Override the IDMItem::sourceEnd member function in the control's derived drag item, if needed, to perform any additional clean-up.

The Workplace Shell print object does not modify the default drag operation. However, you can set a flag in the control's derived drag item, from within the sourcePrint override, to clearly identify the print operation.

You need to implement the rest of the print logic.

Note: If a user drops multiple items on a print object, the Workplace Shell issues a print and an end event for each item.

Enabling a Control for Workplace Shell File Support

Target Support

When a drag is started on an OS/2 Workplace Shell object that represents a text file, the Workplace Shell places information in the following underlying drag items:

- The drag item types
- The RMFs, one of which is IDM::rmFile,IDM::rfText, the file rendering mechanism and text format
- A container name, which will contain the drive and path
- A source name, which will contain the file name

The default target renderer provides support for the file rendering mechanism and text format. To enable a control for Workplace Shell file support, do the following:

1. Override IDMItemProvider::provideEnterSupport in the derived drag item provider class to perform drag item type verification if required.
2. Override IDMItem::targetDrop in the derived drag item class.
3. Obtain the file's drive and path information in the IDMItem::targetDrop override by calling the IDMItem::containerName member function.
4. Obtain the file name in the IDMItem::targetDrop override by calling the IDMItem::sourceName member function.
5. Perform desired processing on the file.

An example of Workplace Shell file support is shown in the drag2 sample that is discussed in "Enabling a Control as a Drop Target" on page 439.

Source Support - Target File Rendering

To enable a control as a file source, the Workplace Shell places the following requirements upon the drag item objects:

- A drag item type

- The file rendering mechanism and text format
- A container name, which will contain the source drive and path
- A source file name
- A suggested target file name

Do the following to implement source file support to allow the Workplace Shell to do target rendering:

1. In the source drag item constructor of the derived drag item class, either add or set an item type of IDM::plainText, using IDMItem::addType or IDMItem::setTypes, respectively.
2. Also, either add or set the file rendering mechanism and text format, <IDM::rmFile,IDM::rfText>, using IDMItem::addRMF or IDMItem::setRMFs, respectively.
3. Set the source drive and path using IDMItem::setContainerName.
4. Set the source file name using IDMItem::setSourceName.
5. Set the suggested target file name using IDMItem::setTargetName.

These steps build the information that the Workplace Shell requires to do target rendering: create the file without further intervention from the source of the drag operation.

Source Support - Delayed or Source File Rendering

If it is not desirable to use target rendering to create the file, you can use what is known as delayed or source file rendering.

For example, you could use delayed file rendering when the file you are dragging requires dynamic modifications. By delaying the file's creation at the target, you have the opportunity to make the dynamic modifications.

Do the following to implement source file support for delayed file rendering:

Note: When the container name is set to 0, which is the default, the Workplace Shell selects the delayed file rendering mechanism.

1. In the source drag item constructor of the derived drag item class, either add or set an item type of IDM::plainText, using IDMItem::addType or IDMItem::setTypes, respectively.
2. Also, either add or set the file rendering mechanism and text format, <IDM::rmFile,IDM::rfText>, using IDMItem::addRMF or IDMItem::setRMFs, respectively.
3. Set the source file name using IDMItem::setSourceName.

4. Set the suggested target file name using IDMItem::setTargetName.
5. Override IDMItem::sourceRender in the derived drag item class.
   a. Perform desired processing on the file in the override.
   b. Call IDMRenderEvent::setCompletion to set the render completion code in the override.
   c. Return from the override.

Setting and Querying the Drag Operation

The default operation for direct manipulation is IDMOperation::drag. The direct manipulation target determines the type of operation (for example, move, copy, or link) based upon the allowable operations defined by the item. However, you can override this setting in a derived item's IDMItem::generateSourceItems function using IDMOperation::setOperation.

Note: The target continually updates this setting, which can be dynamically manipulated using the keyboard augmentation keys. It can be queried using the IDMOperation::operation function.

If the direct manipulation source needs to determine which operation occurred at the target, the operation can be queried using the IDMSourceOperation::operation function. This is sometimes required in a derived item's IDMItem::sourceEnd function override.

For example, you could distinguish a move from a copy operation so you remove the object from the source if you were performing a move operation.

Adding Images to Drag Items

When you drag an object, a visual image is displayed for that object. The User Interface Class Library provides default system images, or you can change the image style and provide your own images.

To change the drag image style, use the IDMSourceOperation::setImageStyle member function. We recommend that you call setImageStyle from the IDMItem::generateSourceItems member function of the application's derived item class.

The following table describes the IDMImage styles and the steps you must take to use them:

IDMImage Style	Description	What to Code
systemImages	If one item is dragged, the ISystemPointerHandle::singleFile icon is used. For more than one item, the ISystemPointerHandle::multipleFile icon is used. Any images supplied with drag items are ignored.	Default
allStacked	Shows each image provided in each drag item. If no images are specified, system images are used.	Attach IDMImage objects to each IDMItem object
stack3AndFade	Shows the first three images provided in the drag items and then shows a special icon that looks like the rest of the images fading out. This is optimal when the user can drag more than three items. If no images are specified, system images are used.	Attach IDMImage objects to three IDMItem objects.

You can use the IDMSourceOperation::setStackingPercentage member function to define the stacking percentage used to calculate the placement of the next stacked image when the image style, IDM::stack3AndFade or IDM::allStacked, is specified. By default, the percentage for both the x and y axis is defined as 50 percent of the current image's size, which results in the placement of the origin, bottom left-hand corner, of the next image in the center of the current image. The placement of the first image is determined by the position of the mouse pointer. The default direction of stacking is toward the upper right. Increase the stacking percentage to expand the stacking of images, and conversely, decrease the stacking percentage to compress the stacking of images. Also, you can alter the direction of stacking using negative percentages as shown below:

```
IDMSourceOperation::setStackingPercentage( IPair( x, y ) );   //stacking direction is upper right
IDMSourceOperation::setStackingPercentage( IPair( -x, y ) );  //stacking direction is upper left
IDMSourceOperation::setStackingPercentage( IPair( -x, -y ) ); //stacking direction is lower left
IDMSourceOperation::setStackingPercentage( IPair( x, -y ) );  //stacking direction is lower right
```

Here, `x` is the stacking percentage for the x-axis and `y` is the stacking percentage for the y-axis.

You can set this function only once per each drag operation.

The drag4 sample contains a simple illustration of the use of the IDMSourceOperation::setStackingPercentage function.

You can attach IDMImage objects to IDMItem objects by using the IDMItem::setImage member function in:

- The constructor of the derived item object
- The implementation of the IDMItem::generateSourceItems member function

The following example adds the text I-beam pointer as an image to a derived IDMItem in its constructor:

```
MyItem::MyItem (IDMSourceOperation* pIDMSrcOp)
{
⋮
 IDMImage image = IDMImage(ISystemPointerHandle(
                            ISystemPointerHandle::text));
 setImage(image);
 }
```

Drag Image Resources for stack3AndFade

When you use the IDMItem::stack3AndFade style, the User Interface Class Library uses a fade icon that looks like the images are fading out. If your application is shipped as a product and uses the stack3AndFade option, you need to ensure its availability to your application.

The fade icon is stored in the User Interface Class Library resource dynamic link library CPPOOR3U.DLL.

If your application is dynamically linked to the User Interface Class Library, follow these steps to use the fade icon:

1. Rename the resource DLL with the DLLRNAME tool shipped with the IBM VisualAge C++ compiler.

   For more information about DDLRNAME, see the *IBM VisualAge C++ Compiler Utilities Reference*.

2. Call ICurrentApplication::setResourceLibrary with the new DLL name as its argument.

   See ICurrentApplication in the *Open Class Library Reference* for more information about setResourceLibrary.

If your application is linked to User Interface Class Library static libraries, follow these steps to use the fade icon:

1. Bind the fade icon to your application .EXE file. The fade icon, fade.ico, and its resource file, DDE4U001.RC, are in the `\ibmcpp\ibmclass` directory on the drive you installed the product on.

2. Call ICurrentApplication::setResourceLibrary with 0 as its argument. The parameter 0 indicates that the fade icon is in the application .EXE file.

   See ICurrentApplication in the *Open Class Library Reference* for more information about setResourceLibrary.

Setting the Target Emphasis

The IDMTargetEvent::presSpace and IDMTargetEvent::releasePresSpace functions are defined to assist IDMTargetEnterEvent, IDMTargetLeaveEvent, and IDMTargetDropEvent events in the drawing and removal of target emphasis. You must use these functions to acquire and release the presentation space that is used to draw target emphasis. IWindow::presSpace and IWindow::releasePresSpace do not work. The drag2 sample contains a simple implementation of target emphasis support.

Debugging Direct Manipulation within an Application

Use the following tips to assist you in using the IBM VisualAge Debugger (ICSDEBUG.EXE) to debug direct-manipulation-enabled applications:

1. Ensure that you are using the proper level of the OS/2 Kernel. The following version of the Kernel must be installed if you are using OS/2 V2.1:

   ```
   OS2KRNL       738648   10-22-93   12:37p
   ```

   This prevents the internal resource interlock problem from occurring. OS/2 V2.11 already includes the fix for this problem.

2. Set the PM debugging mode to Asynchronous and do the following:

   - Install the CSD CTU0002 for Debugger updates if you are using OS/2 V2.1
   - Install the CSD CTU0003 for Debugger updates if you are using OS/2 Warp V3.0

3. Enable the static debug support flag, defined within IDMCOMM.HPP, in your application:

   ```
   IBase::Boolean IDM::debugSupport = true;
   ```

4. The system makes the mouse pointer invisible at certain stages during its processing of the direct manipulation request. To make the mouse visible, use the keyboard to display an OS/2 command prompt and run a utility, such as Petzold’s SHOWPTR.EXE. This utility is available from various electronic media.

You can also use the following tracing options to debug direct-manipulation-enabled applications:

1. Use IMODTRACE_DEVELOP to log the entry and exit of a member function. For example:

   ```
    IMODTRACE_DEVELOP( "CustomerItem::targetDrop" );
   ```

2. Use ITRACE_DEVELOP to log messages, conditions, and variable values. For example:

```
ITRACE_DEVELOP( "Drop failed because xyz function failed; return code was "
                + IString( ulRc ) );
```

Note: Beware of the use of IDMItem::Handle, IDMTargetOperation::Handle, and IDMSourceOperation::Handle with constructor initializers. There is a bug in the IRefCounted class in regard to constructor initializers that makes an application trap, not where the bug occurs, but later in another location. Avoid the following constructor initializer:

```
IDMTargetOperation::Handle TargetOp = IDMTargetOperation::targetOperation();
```

To prevent a trap, break the preceding statement into a declaration and an assignment. The following code effectively bypasses the bug:

```
/**********************************************************/
/* Using IDMItem version of the targetOperation function   */
/**********************************************************/
    IDMTargetOperation::Handle TargetOp;
    TargetOp = targetOperation();
```

Debugging Direct Manipulation

37 Defining Application Resources

Motif The User Interface Class Library for AIX provides a tool, ipmrc2X, to help you convert OS/2 resource (.RC) files to X Toolkit resource files.

For more information about the tool, refer to “Converting Resource Files” on page 463.

Motif AIX does not support OS/2 Presentation Manager (PM) dialog templates. If you write portable applications, use canvases instead of dialog templates. The Hello World version 4 sample application shows you how to do this.

Refer to Chapter 49, “Adding Dialogs and Push Buttons” on page 621 for more information.

PM You define resources in a resource file. A *resource file* is a file that contains data that your application uses, such as text strings and icons. For example, you can define a menu, string table, or dialog template menu or string table in the resource file. You also define the string ID, corresponding to each static string you use in a window, in the resource file.

The resource compiler produces a compiled version of the resources, which is then incorporated into the application’s executable code or stored in a dynamic link library (DLL) for use by one or more applications.

A benefit of defining resources in a resource file is that you can make changes to resource definitions without affecting the application code itself.

You can also provide national language versions by storing the resources for each language in a separate resource file. You can then build your application as separate executable versions for each language (each with a different resource file bound to it) or as a single executable with a separate .DLL for each language.

Refer to “Supporting Double-Byte Character Set and Multiple Languages” on page 464 for more information about multiple language versions.

Using Window Resources

Window resources are read-only data segments stored in an application’s .EXE file or in the dynamic link library’s .DLL file. Predefined User Interface Class Library window resources include keyboard accelerator tables, icons, menus, and bitmaps.

Most window resources are stored in a format that is unique to each resource type. The system translates the formats, as necessary, for use in PM functions.

To access window resources, you must prepare a resource file (ASCII file with the extension .RC). Then the ASCII resource file must be compiled into binary images using the resource compiler. The compiled resource extension is .RES; it can be linked to your application's .EXE file or to a dynamic link library's .DLL file.

Understanding Dialog Templates

A *dialog template* is an OS/2 PM data structure that describes a dialog window and its control windows. PM uses the data in the dialog template to create the dialog window and control windows. An application can create a dialog template at run time, or it can use the system resource compiler to create a dialog-template resource.

Motif AIX does not support OS/2 PM dialog templates. If you write portable applications, use canvases instead of dialog templates. The Hello World 4 sample application show you how to do this.

🕮 Refer to Chapter 49, "Adding Dialogs and Push Buttons" on page 621 for more information.

Accessing Bitmap and Icon Resources

Motif The User Interface Class Library accesses bitmap and icon resources using an IResourceID (an unsigned long). At run-time, IResourceLibrary::loadBitmap or IResourceLibrary::loadIcon must be able to map the IResourceID to the XPM file name. For example:

```
reslib.loadBitmap(42);
```

in Motif causes the User Interface Class Library to search the X resource database for an application resource named *ICLPIXMAP_42. The value is the file name, as follows:

```
*ICLPIXMAP_42: /user/X11/lib/bitmaps/mybitmap.xpm
```

In this case the IResourceId locates a fully qualified XPM file name. Once IResourceId finds the resource that corresponds to a bit-map or icon file name, the User Interface Class Library uses the following search order to locate the XPM file:

1. If the file name begins with a "/", treat it as a fully qualified name. If this file cannot be found, loadBitmap or loadIcon throws an exception.
2. If the name is not fully qualified, then the User Interface Class Library attempts to load the file from the current directory.

3. If the XPM file is not found in the current directory, then the User Interface Class Library attempts to load it from the directory specified by the XAPPLRESDIR environment variable. If the file still cannot be found, loadBitmap or loadIcon throws an exception.

You can create an application resource, such as `*ICLPIXMAP_42` in the previous example by converting an existing PM .RC file using the ipmrc2X tool. If you do not have an existing PM .RC file, create one with a text editor. The advantage here is that you can use preprocessor defines to give the IResourceID's symbolic names and you use the same defines in both your resource files and application.

When ipmrc2X encounters an icon or bit-map resource in the input .RC file, it converts a .BMP or .ICO file name extension to .xpm. Therefore, if your .RC file contains the following lines:

```
ICON    21    test1.ico
BITMAP  22    test2.bmp
```

they convert to the following ipmrc2X output:

```
*ICLPIXMAP_21: test1.xpm
*ICLPIXMAP_22: test2.xpm
```

This allows the PM and Motif versions of the bit maps and icons to exist in the same directory. Remember that ibmp2X converts test2.bmp to test2.xpm. Thus, ipmrc2X and ibmp2X support the same naming scheme.

Note: If you have an User Interface Class Library application for OS/2 that contains bit maps and an icons with the same name, rename one or the other so that your files are not overlaid when you convert them to the AIX format.

Adding Keyboard Accelerators

A *keyboard accelerator* is a keystroke that generates a command message for an application. Using a keyboard accelerator lets a user quickly access commands. It has the same effect as choosing a menu item. While menus provide an easy way to learn an application's command set, accelerators provide quick access to those commands.

The following example shows you how to define keyboard accelerators in a .RC file:

```
ACCELTABLE ID_MENU
{
   "I",  ID_ABOUT_ITEM, CONTROL, SHIFT, ALT
   "I",  ID_ADD_ITEM,   CONTROL, SHIFT
   "a",  ID_ASC-ITEM,   CONTROL
   "d",  ID_DESC_ITEM,  CONTROL
}
```

When a user presses a key sequence of **Ctrl+Shift+Alt+I**, this causes an ID_ABOUT_ITEM event. A **Ctrl+Shift+I** key sequence causes an ID_ADD_ITEM command event. Notice that the more restrictive accelerator is coded first in the previous example.

Without accelerators, a user might generate commands by pressing the **Alt** or **PF10** keys to access the menu bar, using the **Arrow** keys to select the item, and then pressing the **Enter** key to choose the item. In contrast, accelerators let users generate commands with fewer keystrokes.

Although, normally, accelerators are used to generate existing commands as menu items, they also can send commands that have no menu-item equivalent.

Understanding Accelerator Tables

An accelerator table contains an array of accelerators. Accelerator tables exist at two levels within the operating system: a single accelerator table for the system queue and individual accelerator tables for application windows.

Accelerators in the system queue apply to all applications. For example, the **F1** key always generates a help message.

Having accelerators for individual application windows ensures that an application can define its own accelerators without interfering with other applications. An accelerator for an application window can override the accelerator in the system queue.

An application can modify both its own accelerator table and the system's accelerator table.

Creating an Accelerator-Table Resource

You can use an accelerator in an application by creating an accelerator-table resource in a resource-definition file. Then, when the application creates a standard frame window, the application can associate that client menu bar with the resource.

As specified in a resource-definition file, an accelerator table consists of a list of accelerator items, each defining the keystroke that triggers the accelerator, the command the accelerator generates, and the accelerator's style. The style specifies whether the keystroke is a virtual key, a character, or a scan code, and whether the generated message is the default.

Converting Resource Files

Motif

To use the ipmrc2X tool to convert OS/2 .RC files to Motif resource files, use the following code:

```
ipmrc2X [-I<include path>] [-A<Application Class>] input [output]
```

include path

The directory or directories that the preprocessor looks in for the include files. If you do not specify this, all include files you reference in the input file must be in the current directory.

Application Class

The application class that is assigned to the created X resources. If you do not supply this, a value of “*” is used.

input

A file containing PM style resources; for example, a .RC file.

output

The output file name. If you do not supply the output file name, the output is written to the current directory without the .rc extension and the first one or two characters of the file name appear in uppercase letters. (The first two characters are capitalized when the file name starts with an “x”.)

The following table shows an example of the input and output files:

Input	Output
myapp.rc	Myapp
xedit.rc	XEdit

The following example shows the ipmrc2X command to use to convert the myapp.rc file:

```
ipmrc2X -I/usr/include/myapp -AMyapp myapp.rc /u/dev1/defaults/Myapp
```

Working with the ipmrc2X Tool

As you work with ipmrc2X, keep in mind the following:

- It handles the following types of resources:

  accelerator table (ACCELTABLE)
  bit map (BITMAP)
  help tables (HELPTABLE)
  help subtable (HELPSUBTABLE)
  icon (ICON)
  menu (MENU)

menu item (MENUITEM)
string table (STRINGTABLE)
submenu (SUBMENU)

- The accelerator table, menu, menu item, string table, and submenu definitions cannot span multiple lines. For example, the ipmrc2X tool cannot handle the following code; however, the resource compiler on OS/2 PM can.

```
STRINGTABLE
BEGIN
  STR_HELLO,
    "Hello World"
END
```

- ipmrc2X error checking does not flag the same errors as the OS/2 resource compiler does. For example, incomplete lines and missing commas produce erroneous output from ipmrc2X but no error message. When you use the imprc2X check input files for errors using the OS/2 resource compiler.
- ipmrc2X, like the OS/2 resource compiler, processes only a single input file. For this reason, do not use wildcard (*) characters in the input file name.
- Place the converted resource file in your $HOME directory or point to it by using the XAPPLRESDIR environment variable.

Supporting Double-Byte Character Set and Multiple Languages

You can use one source file for your application code and then provide the double-byte character set (DBCS) and support multiple languages by using separate resource files for each of the languages you support. The User Interface Class Library approach includes either of the following:

- Use a single executable file with a separate .DLL for each language.
- Use separate executable files for each language (each with a separate resource file bound to it).

Creating DBCS-Enabled Applications

The following suggestions can assist you in creating DBCS-enabled applications:

- Use the canvas classes to build dialogs because you define message strings in resource files, you can translate them easily to another language without changing the source code.
- Use the IEditVerifyHandler class, which provides all the keyboard action event information, for DBCS-enabled applications. To process both single- and double-byte character key events, use the mixedCharacter member function. To process only single-byte characters, use the character member function.

- Use the IString class, which is DBCS-enabled and supports mixed strings that contain both the single-byte character set (SBCS) and DBCS characters. Objects of the IString class are essentially arrays of characters. The IString class provides functions to test the characters that make up the string. These functions help users determine whether a character is single byte or multiple byte, and whether it is a valid DBCS first byte.
- Use the IDBCSBuffer class, which ensures that the search functions do not inadvertently match the second byte of a DBCS character. The IDBCSBuffer class is derived from the IBuffer class, which holds the IString contents. The two bytes of a DBCS character will not be split.
- Use the following member functions in a DBCS-enabled application:

Member Function	Returns True If...
isCharValid	The character at the given index is in the set of valid characters
isDBCS1	The byte at the given offset is the first byte of DBCS
isPrevDBCS	The character preceding the one at the given offset is a DBCS character

- Specify one of the following data type styles when you create and manage the IEntryField and IComboBox classes:

Data Type Styles	Allows the Following Input...
anyData	A mixture of SBCS and DBCS characters.
dbcsData	DBCS-only data.
mixedData	A mixture of SBCS and DBCS characters. Use this style if you plan to convert data to an EBCDIC code page.
sbcsData	SBCS-only data.

- Specify the appDBCSStatus style when constructing an IFrameWindow to include a DBCS status area when the frame appears in a DBCS environment. The User Interface Class Library automatically shares DBCS status control between a parent and child frame window.

Multiple Language Support

38 Adding Events and Event Handlers

The User Interface Class Library uses events and event handlers to encapsulate the message architecture of OS/2 Presentation Manager (PM) in an object-oriented way. The User Interface Class Library reserves message IDs beginning at 0xFFE0. If you use the User Interface Class Library, define user messages only in the range of WM_USER (0x1000) through 0xFFDF.

Figure 60 shows the relationships between window, event, and handler classes.

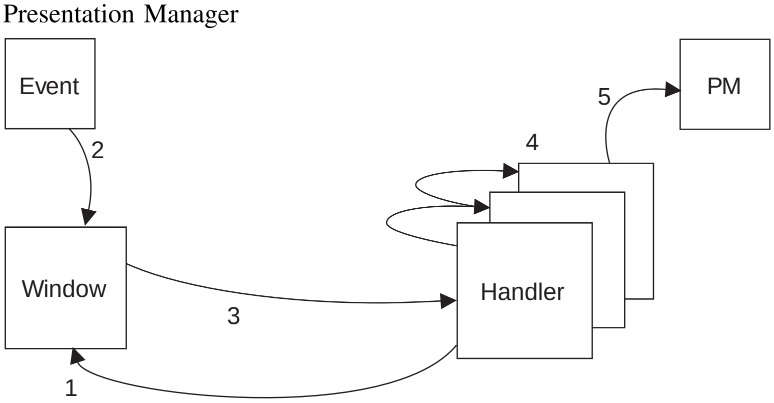


Figure 60. Relationship of Window, Event, and Handler Classes to

1. Handlers are registered with the window.
2. PM messages are encapsulated in event objects, which are passed to the window or control that had the event.
3. The window then invokes the handlers attached to it, passing the event object as a parameter.
4. The handlers are called sequentially with the most recently added handler invoked first. A handler indicates when processing for the event is complete by returning a Boolean value of true.
5. If none of the handlers process the event, it is passed up the owner chain.

The distinction between window classes and handler classes lets you separate the event-handling logic from the rest of the application. This enables reuse of this logic. For example, you can reuse a handler to verify telephone numbers wherever an entry field accepts telephone numbers.

Processing Events Using Handlers

Each handler class has one or more virtual functions that are called to process the event. When an application processes events, it normally subclasses a handler class and overrides the virtual function to provide its own application-specific logic.

When you are within a handler member function, do not delete the IWindow object to which the handler is attached.

The order in which you attach handlers can cause one not to receive events, because handlers are called in the reverse order that they are attached.

Ensure that handlers return from virtual functions within 1/10 second to avoid locking up the system by delaying the PM message processing.

Figure 61 shows how the ICommandHandler works. All handler classes contain a dispatchHandlerEvent function to determine whether the handler needs to process the event or return it. If the event needs processing, it creates the appropriate event object and calls the appropriate virtual function to process the event.

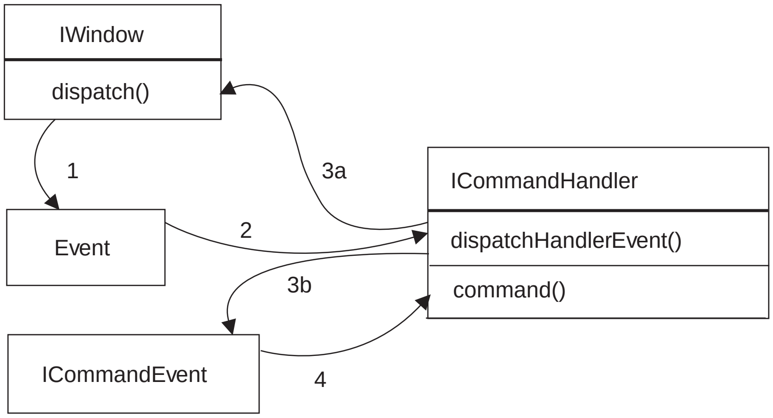


Figure 61. Processing within the ICommandHandler

The numbers in Figure 61 represent the following:

Number	Description
1	Window creates an event.
2	IEvent is passed to ICommandHandler.
3a	IEvent is not processed.
3b	ICommandEvent is generated.
4	ICommandEvent is processed by command().

Table 12 on page 470 presents some common events for which you can provide handlers.

Note: This is not a complete list of all events and their handlers.

It relates the type of event, the handler for that event, and the member function in the handler class that the application must override to provide its own logic.

The *Open Class Library Reference* contains descriptions of all handler classes and member functions.

Table 12 (Page 1 of 2). Common Events and Their Handlers

Event Generated by	Event Class	Handler Class	Member Function	PM Message
Command event by menu selection, push button, or accelerator key	ICommandEvent	ICommandHandler	command	WM_COMMAND
System command event by menu selection, push button, or accelerator key	ICommandEvent	ICommandHandler	systemCommand	WM_SYSCOMMAND
Edit event by entry field, combination box, MLE, or slider	IControlEvent	IEditHandler	edit	WM_CONTROL
Gain focus or lose focus by entry field, combination box, MLE, slider, container, or spin button	IControlEvent	IFocusHandler	getFocus, lostFocus	WM_CONTROL
Keyboard entry by entry field, combination box, MLE, or other input focus control	IKeyboardEvent	IKeyboardHandler	keyPress, key, scanCodeKeyPress, virtualKeyPress, characterKeyPress	WM_CHAR
Paint area event by all controls	IPaintEvent	IPaintHandler	paintWindow	WM_PAINT
Resize event by all controls	IResizeEvent	IResizeHandler	windowResize	WM_WINDOWPOSCHANGED
Item selected by list box, combination box, container, check box, or radio button	IControlEvent	ISelectHandler	selected	WM_CONTROL
Enter pressed when item selected, or double-click on item by list box, combination box, or container	IControlEvent	ISelectHandler	enter	WM_CONTROL
Pop-up menu requested by mouse button or keyboard	IMenuEvent	IMenuHandler	makePopUpMenu	WM_CONTEXTMENU
Menu about to be shown by pull-down menu or pop-up menu	IMenuEvent	IMenuHandler	menuShowing	WM_INITMENU

Table 12 (Page 2 of 2). Common Events and Their Handlers

Event Generated by	Event Class	Handler Class	Member Function	PM Message
Menu item highlighted and ready to be selected by mouse or keyboard	IMenuEvent	IMenuHandler	menuSelected	WM_MENUSELECT
Menu removed by mouse or Esc key	IMenuEvent	IMenuHandler	menuEnded	WM_MENUEND
Container item context menu requested by container	IMenuEvent	ICnrMenuHandler	makePopupMenu	WM_CONTROL
Process Mouse Events	IMouseEvent	IMouseHandler	mouseClicked, mouseMoved, mousePointerChange	WM_BUTTON1DOWN, WM_BUTTON1UP, WM_BUTTON1DBLCLK, WM_BUTTON2DOWN, WM_BUTTON2UP, WM_BUTTON2DBLCLK, WM_BUTTON3DOWN, WM_BUTTON3UP, WM_BUTTON3DBLCLK, WM_BUTTON1CLICK, WM_BUTTON2CLICK, WM_BUTTON3CLICK, WM_CHORD, WM_MOUSEMOVE, WM_CONTROLPOINTER

Extracting Information from Events

The IEvent class acts as the base class for the more specialized event classes. It provides general member functions to extract the message ID and message parameters. The subclasses of IEvent generally add more specialized functions for extracting information specific to that type of event.

Table 13 shows some common event classes and some of the functions they contain to extract event information.

Table 13. Event Classes and Accessor Functions

Event Class	Accessor Function	Description of Return Value
IEvent	window	The IWindow object pointer
IEvent	handle	IWindowHandle of the window
IEvent	eventId	ID of the event
IEvent	parameter1	IEventData containing first event parameter
IEvent	parameter2	IEventData containing second event parameter
ICommandEvent	source	An enumeration type that gives the type of control
ICommandEvent	commandId	The ID of the command that caused the event
IControlEvent	controlId	The ID of the control that caused the event
IControlEvent	control	Pointer to the control that caused the event
IKeyboardEvent	character	Single-byte character code (exception thrown if DBCS)
IKeyboardEvent	mixedCharacter	IString containing character (can be DBCS)
IKeyboardEvent	virtualKey	An enumeration type that gives the virtual key event
IMenuEvent	menuItemId	The ID of the selected menu Item
IMenuEvent	mousePosition	Position of mouse at the time the event occurred
IPaintEvent	presSpaceHandle	The handle of the presentation space to use for any drawing
IPaintEvent	rect	The screen rectangle that needs updating

The IEvent class provides a member function, setResult, for those events that require a value to be returned.

Refer to the *Open Class Library Reference* for a complete list of event classes and member functions.

Writing an Event Handler

In general, writing an event handler can be divided into the following steps:

1. Determine which handler class processes the event.
2. Subclass the handler class and override the event handling functions.
3. Create an instance of your subclass.
4. Start processing events for the window.
5. Stop processing events for the window.

The Hello World application has several event handlers. The following example illustrates how to use the above steps to process user menu selections. The code shown is from Hello World version 3.

1. Determine which handler class processes the event.

   When you select a menu item, an ICommandEvent is generated. The handler class for this type of event is ICommandHandler.

2. Subclass the handler class and override the event-handling function.

   The Hello World application creates a new class called ACommandHandler that is derived from the ICommandHandler class. The virtual function, ICommandHandler::command processes command events. The class ACommandHandler overrides this function to provide its own command event handling.

   The following sample, taken from the AHELLOW3.HPP, file, shows the class declaration of ACommandHandler.

   ```
   ⋮
   class ACommandHandler : public ICommandHandler {
   public:
     ACommandHandler(AHelloWindow *helloFrame);

   protected:
   ⋮
   virtual Boolean
     command(ICommandEvent& cmdEvent);

   private:
     AHelloWindow *frame;
   };
   ⋮
   ```

   The public constructor and private data member frame save a pointer to the frame window for which commands will be processed.

   The ACommandHandler command function provides command processing for AHelloWindow class objects. The definition of the command function is taken from AHELLOW3.CPP. The ID of the menu item is extracted from the command event object using the commandId member function.

```
⋮
IBase::Boolean
  ACommandHandler :: command(ICommandEvent & cmdEvent)
{
  Boolean eventProcessed(true);         //Assume event will be processed
⋮
  switch (cmdEvent.commandId()) {
    case MI_CENTER:
      frame->setAlignment(AHelloWindow::center);
      break;
    case MI_LEFT:
      frame->setAlignment(AHelloWindow::left);
      break;
    case MI_RIGHT:
      frame->setAlignment(AHelloWindow::right);
      break;

    default:                            //Otherwise,
      eventProcessed=false;             //  the event wasn't processed
  } /* end switch */

  return(eventProcessed);
```

3. Create an instance of your subclass.

   Define a data member from your new handler class in your application window. The following code comes from the AHELLOW3.HPP file.

```
⋮
    ACommandHandler commandHandler;
⋮
```

   You should also add an initializer to the constructor for the application window. This is shown in the AHELLOW3.CPP file.

```
⋮
AHelloWindow :: AHelloWindow(unsigned long windowId)
  : IFrameWindow(IFrameWindow::defaultStyle() |
                 IFrameWindow::minimizedIcon,
                 windowId)
   ,menuBar(WND_MAIN, this)
   ,statusLine(WND_STATUS, this, this)
   ,hello(WND_HELLO, this, this)
   ,infoArea(this)
   ,commandHandler(this)
⋮
```

4. Start processing events for the window.

   The base class IHandler provides a member function handleEventsFor to attach a handler to a window. In the Hello World application, AHELLOW3.CPP, the ACommandHandler begins processing command events for the AHelloWindow in its constructor with the following statement:

```
⋮
commandHandler.handleEventsFor(this);
⋮
```

5. Stop processing events for the window.

   The base class IHandler provides a member function stopHandlingEventsFor to stop event processing for the window. In the Hello World application, AHELLOW3.CPP, the ACommandHandler stops processing command events for the AHelloWindow in its destructor with the following statement:

   ```
   ⋮
   commandHandler.stopHandlingEventsFor(this);
   ⋮
   ```

Extending Event Handling

The User Interface Class Library provides handlers for common Presentation Manager (PM) messages. However, you may find it necessary to process messages for which there are no predefined handler classes. The User Interface Class Library makes it easy to add new event and handler classes.

The IHandler class is designed to act as a base class for handlers. All event handlers are derived from this class.

The following statements from the ATIMEHDR.CPP file of Hello World version 6 show a way to provide a new handler class derived from IHandler. This sample uses timer functions to implement a timer event handler.

Note: The ATimeHandler class demonstrates IHandler derivation; the timer functions might not handle all cases and might not work in a multithreaded environment.

The steps for creating an IHandler class, ATimeHandler, follow.

Note that because the timer functions on the AIX and OS/2 operating systems are different. You can use ifdef compiler directives to determine which functions to call. This allows the application to be portable.

This is just an example of creating handlers. The User Interface Class Library contains classes you can use to set time intervals.

See "Setting Time Intervals" on page 591 for more information about the ITimer class.

1. Subclass the IHandler class by creating a class declaration for ATimeHandler. The class is derived from IHandler and provides a virtual function tick to process the event. The following code comes from the ATIMEHDR.HPP file:

```
class ATimeHandler : public IHandler
{
typedef IHandler
  Inherited;
public:
  ATimeHandler() : timerId(0) { }       //Initialize timerId data member

virtual ATimeHandler
  &handleEventsFor(IWindow *window),
  &stopHandlingEventsFor(IWindow *window );
protected:
⋮
Boolean
  dispatchHandlerEvent(IEvent& event);
virtual Boolean
  tick(IEvent& event);
⋮
private:
  unsigned long timerId;
};
```

2. Override the handleEventsFor member function.

   This function starts the handler. In the Hello World version 6 ATIMEHDR.CPP file, the first timer starts, using a constant time interval of 1 second, as follows:

```
ATimeHandler
  &ATimeHandler :: handleEventsFor( IWindow *window )
{

#ifdef IC_MOTIF
⋮
  timerId = XtAppAddTimeOut (
                 XtWidgetToApplicationContext ((Widget)window->handle()),
                 timeInterval,
                 (XtTimerCallbackProc) postATimeHandlerEvent,
                 window);
#endif

#ifdef IC_PM
⋮
  timerId = TIMER_ID;
  WinStartTimer( IThread::current().anchorBlock(),
               window->handle(), timerId, timeInterval);
#endif
⋮
  Inherited::handleEventsFor(window);
  return (*this);
} /* end ATimeHandler :: handleEventsFor(...) */
```

   The “typedef IHandler Inherited” statement in the constructor lets you generically call inherited functions that you have overridden. In this sample, the handleEventsFor function from IHandler is called to complete the starting of the handler.

3. Optionally, post the event.

The PM timer function automatically posts a WM_TIMER event to the window specified as the second argument of the timer call. In this case, you do not have to provide any additional processing to post the event.

In contrast, the AIX timer function uses a callback method for notifying the application when the timer has expired, but it does not post an event. Therefore, the callback routine must do the posting. The function to call back is specified as the third argument in the add timer call. This function must be declared as an extern void _System. The function posts the timer event to the window specified in the last argument of the add time-out call. The postTimeHandlerEvent function, from the ATIMEHDR.CPP file, as follows:

```
⋮
  extern void _System                  //Forward declare for post function
    postATimeHandlerEvent (IWindow *, XtIntervalId *);
⋮
    IEventParameter2 newTimer = XtAppAddTimeOut (
              XtWidgetToApplicationContext((Widget)window->handle()),
              timeInterval,
              (XtTimerCallbackProc)postATimeHandlerEvent,
              window);
⋮
    window->postEvent (WM_TIMER, IEventParameter1(*timerUp), newTimer);
  }
} /* end extern void _System postATimeHandlerEvent(...) */
#endif
⋮
```

4. Override the dispatchHandlerEvent member function.

This function determines the relevance of the message. If the message is not relevant, the function returns false and passes the message to other handlers attached to the window. If this event is relevant, then the handler's function for processing the event should be called. In the Hello World version 6 ATIMEHDR.CPP file, the tick function is called, as follows:

```
⋮
IBase::Boolean
  ATimeHandler :: dispatchHandlerEvent(IEvent& event)
{
  Boolean eventProcessed(false);        //Assume event will not be proccessed
⋮
  if ((event.eventId() == WM_TIMER) && (event.parameter1() == timerId))
  {
#ifdef IC_MOTIF
⋮
   timerId = event.parameter2();
#endif
⋮
    eventProcessed = tick(event);
  }
  return (eventProcessed);
} /* end ATimeHandler :: dispatchHandlerEvent(...) */
⋮
```

Because the OS/2 timer is continuous, the timer ID can be a constant number. However, for AIX, a new timer is created every second. Therefore, when the expired timer ID is relevant, for example, dispatched, the new timer ID replaces the old one.

5. Create the member function for processing the event.

   Normally, the event processing function of a general handler class does nothing but return false. It is the specific handler class, AHelloTimeHandler, in Hello World version 6 ATIMEHDR.CPP file that overrides the event processing function and returns true.

```
⋮
IBase::Boolean
  ATimeHandler :: tick(IEvent& event)
{
  return (false);                      //The timer event is not processed
} /* end ATimeHandler :: tick(...) */
⋮
```

   The ATimeHandler::tick member function, in the AHELLOW6.CPP file, overrides the event handling, as follows:

```
⋮
IBase::Boolean
  AHelloTimeHandler::tick(IEvent& evt)
{
  ((AHelloWindow *)evt.window())->tickTime();
  return (true);                       //Event is always processed
} /* end AHelloTimeHandler :: tick(...) */
⋮
```

6. Override the stopHandlingEventsFor member function.

   In Hello World version 6, the timer is removed or stopped, depending on the system, and the inherited stopHandlingEventsFor function completely stops the timer. The following code comes from the ATIMEHDR.CPP file:

```
ATimeHandler
  &ATimeHandler :: stopHandlingEventsFor( IWindow *window )
{

#ifdef IC_MOTIF
⋮
  XtRemoveTimeOut (timerId);
  timerId = 0;
#endif

#ifdef IC_PM
⋮
  WinStopTimer( IThread::current().anchorBlock(),
                window->handle(), timerId);
#endif
⋮
  Inherited::stopHandlingEventsFor(window);
```

```
    return (*this);
  } /* end ATimeHandler :: stopHandlingEventsFor(...) */
```

Refer to Hello World version 6 sample applicaton (AHELLOW6.CPP, AHELLOW6.HPP, AEARTHW6.CPP, and AEARTHW6.HPP files) to see how to derive from ATimeHandler to provide a ticking clock and twinkling stars.

Understanding More About Writing Handlers

To prevent ATimeHandler users from having to understand how information is encoded in the two message parameters inside the event, derive an event class from IEvent to encapsulate this information. The following statements show an example of how to do this:

```
class ATimerEvent : public IEvent
{
public:
  ATimerEvent( IEvent &evt ) : IEvent( evt ) {;}  // Define functions inline

  unsigned long
    timerNumber() const { return parameter1().number1(); }
};
```

You can only construct objects of this class from an instance of IEvent. Because of the small amount of code required, the example defines the code inline.

To use the new class, change the dispatchHandlerEvent member function to create an instance of ATimerEvent. Also, change the ATimeHandler::tick member function to accept an ATimerEvent object as a parameter, as shown in the ATIMEHDR.CPP file:

```
⋮
IBase::Boolean
  ATimeHandler :: dispatchHandlerEvent(IEvent& event)
{
  Boolean eventProcessed(false);        // Assume event will not be processed
⋮
  if ((event.eventId() == WM_TIMER) && (event.parameter1() == timerId))
  {
#ifdef IC_MOTIF
⋮
   timerId = event.parameter2();
#endif
⋮
    eventProcessed = tick(event);
  }
  return (eventProcessed);
} /* end ATimeHandler :: dispatchHandlerEvent(...) */
⋮
IBase::Boolean
  ATimeHandler :: tick(IEvent& event)
{
  return (false);                       //The timer event is not processed
} /* end ATimeHandler :: tick(...) */
⋮
```

The two classes now completely encapsulate timer messages. Users of the classes do not need to know which messages are generated or how the information is encoded in the message parameters.

You can restrict the window classes to which a handler can be attached. The following steps show you how to restrict the attachment of the ATimeHandler class to the ITextControl class and its derived classes.

1. Write the class declaration following this example:

```
class ATimeHandler : public IHandler
{
public:
  /* use default constructor */

Boolean
    dispatchHandlerEvent( IEvent& evt );

virtual ATimeHandler
    &handleEventsFor       ( ITextControl* textWindow ),
    &stopHandlingEventsFor ( ITextControl* textWindow );

protected:
virtual Boolean
    tick( ATimerEvent& evt );

private:                                  //Make these functions private
virtual ATimeHandler                      // so they cannot be called
    &handleEventsFor       ( IWindow* window ),
    &stopHandlingEventsFor ( IWindow* window );
};
```

2. Override the handleEventsFor member function to accept only ITextControl objects, as shown in the following example:

```
⋮
ATimeHandler
  &ATimeHandler::handleEventsFor( ITextControl* textWindow )
{
⋮
  return (handleEventsFor(window));
}
```

3. Override stopHandlingEventsFor member function to accept only ITextControl objects. For example:

```
ATimeHandler
  &ATimeHandler::stopHandlingEventsFor( ITextControl* textWindow )
{
⋮
  return (stopHandlingEventsFor(window));
}
```

Handling Mouse Events

Pointer devices give users the ability to perform actions directly. The User Interface Class Library offers classes to handle the mouse pointer.

You can use IMouseHandler to process a variety of mouse events. These events include button presses and releases, double-clicks, multiple button presses, and mouse moves. You can also query keyboard state information at the time a mouse event is generated.

Begin by creating an IMouseHandler object and then attach it to any kind of window (for example, IMultiLineEdit or ISetCanvas). Although the window that the mouse is over receives a mouse event first, events are sometimes passed on for additional processing to their owner windows. A mouse event continues to travel up the owner window chain until either a handler stops it or the event is processed by the window itself. The mouse handler must return *true* to stop any additional processing of a mouse event.

When an IMouseHandler object receives a mouse event, it creates either an IMouseEvent, an IMouseClickEvent, or an IMousePointerEvent and routes it to a mouse handler virtual function. The mouse handler virtual functions are as follows:

Function	**Purpose**
mouseClicked	Processes a mouse click event.
mouseMoved	Processes a mouse move event.
changeMousePointer	Changes the pointer when the mouse is over the handled window. If you need to change the mouse pointer for a frame window and all its children, use IFrameWindow::setMousePointer.

Whenever a mouse button's state changes, the IMouseHandler calls its mouseClicked function. The IMouseClickEvent object identifies the button, its current keyboard state, and the mouse pointer position. The IMouseClickEvent defines the following virtual functions:

Function	**Purpose**
mouseButton	Returns the clicked mouse button (button1, button2, button3).
mouseAction	Returns the mouse action (clicked, double-clicked).
mouseNumber	Returns the mouse button that changed state (button1, button2).

windowUnderPointer	Returns the handle of the window that the mouse pointer is over.

Note: On a two-button mouse, button1 is the left mouse button on a right-handed mouse and the right button on a left-handed mouse. Button2 is the right mouse button on a right-handed mouse and the left button on a left-handed mouse.

The IMouseEvent defines the following virtual functions:

Function	**Purpose**
windowUnderPointer	Returns the handle of the window that is under the mouse pointer.
isAltKeyDown	Returns true if the Alt or menu key is down when the mouse is moved.
isCtrlKeyDown	Returns true if the Ctrl key is down when the mouse is moved.
isShiftKeyDown	Returns true if the Shift key is down when the mouse is moved.

You initiate a mouse pointer event when you enter and exit a window with a mouse handler attached. The IMousePointerEvent defines the following virtual functions:

Function	**Purpose**
defaultMousePointer	Returns the default mouse pointer for the window that is under the mouse.
setMousePointer	Sets the pointer to use for the window that is under the mouse.
windowId	Returns the window ID of the control that the event applies to.

Mouse Handler Example

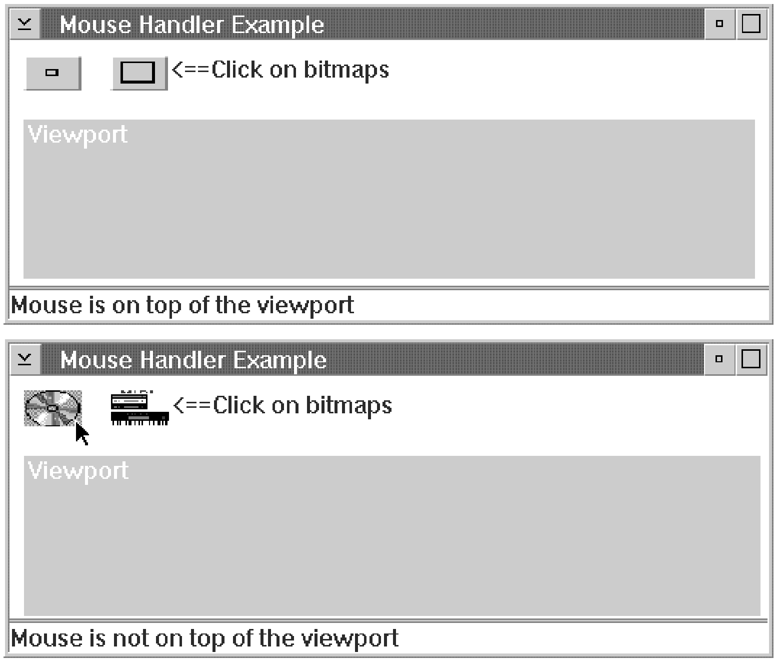
The following code creates a multi-cell canvas as a client window with a view port control and two bitmaps as its children. A mouse handler is attached to the client canvas, the view port and the bitmaps. Code, in the overloaded changeMousePointer member function, changes the mouse pointer when the mouse is over the view port. The information area at the bottom of the frame is updated to indicate when the mouse is over the view port and when it leaves the view port. When the mouse is over the bitmaps, the program invokes the mouseClicked member function. The code in the .hpp file is as follows:

```
/*************************************************************/
/* Define the Mouse Handler                                  */
/*************************************************************/
class AMouseHandler : public IMouseHandler
 {
public:
   AMouseHandler(MainWindow *aFrame);

protected:
   virtual Boolean mouseClicked           (IMouseClickEvent & event);
   virtual Boolean mousePointerChange    (IMousePointerEvent&  event);

private:
   MainWindow * frame;
};


/*************************************************************/
/* Define the Main Window                                    */
/*************************************************************/
class MainWindow : public IFrameWindow {
public:
  MainWindow( unsigned long windowId);
  Boolean handleClickEvent(unsigned long id);
  Boolean handleChangeEvent(IMousePointerEvent& event);

private:

   IMultiCellCanvas    clientCanvas;
   IViewPort           aviewport;
   IStaticText         viewText;
   IStaticText         bmpText;
   IStaticText         infoText;
   IBitmapControl      bmp1;
   IBitmapControl      bmp2;
   IPointerHandle      ptr_bmp;
   AMouseHandler       mouseHandler;
  };
```

The code in the .cpp file is as follows:

```
/*************************************************************/
/* Create the Main Window                                    */
/*************************************************************/
MainWindow::MainWindow( unsigned long windowId)
          : IFrameWindow("Mouse Handler Example", windowId),
                     clientCanvas(REMOTECANVASID, this, this),
                     aviewport(VPID, &clientCanvas, &clientCanvas),
                     viewText(VTXT, &aviewport, &aviewport, IRectangle(),
                       IStaticText::defaultStyle() | IStaticText::center),
                     bmpText(BMPTXT, &clientCanvas, &clientCanvas, IRectangle(),
                       IStaticText::defaultStyle() | IStaticText::center
                        | IStaticText::top),
                     infoText(INFOID, this, this, IRectangle(), IStaticText::defaultStyle()
                        | IStaticText::top
                        | IStaticText::left
                        | IStaticText::wordBreak ),
                     bmp1(BMP1ID,&clientCanvas,&clientCanvas,
                                ISystemBitmapHandle::minimizeButton),
                      bmp2(BMP2ID,&clientCanvas,&clientCanvas,
                                ISystemBitmapHandle::maximizeButton),
                      mouseHandler(this)


 {
     addExtension(&infoText, IFrameWindow::belowClient,
         IFont(&infoText).maxCharHeight(),
         IFrameWindow::thickLine);
     infoText.setText(ID_TEXT);
     bmpText.setText("<==Click on bitmaps");
     bmpText.setForegroundColor(IColor::white);
     viewText.setText(VTXT);
     setClient(&clientCanvas);
     clientCanvas.setBackgroundColor(IColor::blue);
     aviewport.setBackgroundColor(IColor::pink);
     clientCanvas.addToCell(&aviewport,   2,  6, 40, 1);
     clientCanvas.addToCell(&bmp1,  2,  2, 4, 2);
     clientCanvas.addToCell(&bmp2,  8,  2, 4, 2);
     clientCanvas.addToCell(&bmpText, 12, 2, 5, 2);

     ISize size = clientCanvas.minimumSize();
     IPoint point = position();
     moveSizeToClient(IRectangle(point.x(),
                                 point.y(),
                                 point.x() + size.width(),
                                 point.y() + size.height()));

     /*******************************************************/
     /* Add the Mouse handler                               */
     /*******************************************************/
     mouseHandler.handleEventsFor(&aviewport);
     mouseHandler.handleEventsFor(&bmp1);
     mouseHandler.handleEventsFor(&bmp2);
     mouseHandler.handleEventsFor(&clientCanvas);

     IResourceLibrary reslib;
     ptr_bmp = reslib.loadPointer(PTR_BITMAP);

     show();
     setFocus();
}

/************************************************************/
/* Change the mouse pointer and update the info area        */
/************************************************************/
IBase::Boolean MainWindow::handleChangeEvent(IMousePointerEvent& event)
{
  if (event.windowId() == IC_VIEWPORT_VIEWRECTANGLE)
    {
    event.setMousePointer(ptr_bmp);
    infoText.setText("Mouse is on top of the viewport");
    return true;
    }
  else
    {
    infoText.setText("Mouse is not on top of the viewport");
    return false;
    }

}

/*************************************************************/
/* Change the button bitmaps and update the info area.       */
/*************************************************************/
IBase::Boolean MainWindow::handleClickEvent(unsigned long id)
{

  switch (id)
   {
    case BMP1ID:
      bmp1.setBitmap(BMPCD);
      infoText.setText("minimize button pushed");
      break;
    case BMP2ID:
      bmp2.setBitmap(BMPMIDI);
      infoText.setText("maximize button pushed");
      break;
    }
  return true;
}


AMouseHandler :: AMouseHandler(MainWindow *aFrame)
{
   frame=aFrame;
}

/*************************************************************/
/* Hande the mouse click event                               */
/*************************************************************/
IBase::Boolean  AMouseHandler :: mouseClicked (IMouseClickEvent & event)
{
   IWindow * pWindow =IWindow::windowWithHandle(event.windowUnderPointer());
   unsigned long int winId = pWindow->id();
   frame->handleClickEvent(winId);
   return true;
}

/*************************************************************/
/* Hande the mouse pointer change event                      */
/*************************************************************/
IBase::Boolean AMouseHandler :: mousePointerChange (IMousePointerEvent& event)
{

  return (frame->handleChangeEvent(event));
}
```

Figure 62 shows the results of using the mouse handler.



Mouse is on top of the viewport

Mouse is not on top of the viewport

Figure 62. Results of Using the Mouse Handler

39

Understanding Fonts

The IFont class contains member functions to set and change the characteristics of the fonts you use in your applications. You can set the font of IWindow objects using the member function setFont, which is defined in the IWindow class.

Constructing Fonts

There are two ways you can construct IFont objects in your application:

- Create IFont objects with a specific font face name
- Create IFont objects using a window's font

Creating an IFont Object with a Specific Name

The highlighted lines in the following example show you how to create a font with a specific name and point size, and then how to change the point size of the text associated with visible User Interface Class Library objects.

For portable applications, consider using a font dialog. For more information about font dialogs, refer to "Creating a Font Dialog" on page 385.

Note: Font names change depending on the system you use.

```
#include <ifont.hpp>
⋮
  IFont font("Helvetica",8);
⋮
  title1.setAlignment( IStaticText::centerLeft );
  title1.setText( STR_TITLE1 );
  font.setPointSize(12);
  title1.setFont(font);
⋮
  check1.setText( STR_CHECK1 );
  font.setPointSize(20);
  check1.setFont(font);
⋮
```

To test the font statements, include the highlighted lines in the AMCELCV.CPP file. The AMCELCV.CPP file is located in the `\ibmcpp\samples\ioc` directory.

Creating an IFont Object Using a Window's Font

The second way to create an IFont object is to use an existing window's font. This constructs a font object with the same name that is used in the window. If no window is specified, the system default font is used.

Fonts

The following example sets the height of an information area using the maximum character height of the font used for the information area.

```
setExtensionSize(&infoArea, IFont(&infoArea).maxCharHeight());
```

📖 Refer to the *Open Class Library Reference* for more information on the IFont class.

Adding Clipboard Support

A *clipboard* is a systemwide place for users to store data temporarily. The clipboard enables your user to move data within a single application or to exchange data among applications. Typically, a user selects data in the application using the mouse or keyboard, then initiates a cut or copy operation on the selected data. The clipboard can hold an entire object or part of that object, and it can hold any kind of object. For example, the clipboard can hold a single line of text or an entire database, a single line segment or an entire graphic.

When the user selects the paste operation, the data is transferred to the application from the clipboard.

Note: Only a single item of data can be stored in the clipboard at a time. Therefore, do not use the clipboard to store data unless a user requests it because you can overlay the user's data stored there. This is important: the user must always control access to the clipboard.

While you can only store a single item of data in the clipboard, you can store this item in multiple formats. This allows an application to choose the format it supports that gives it the most information about the data. For example, a graphics application might copy a picture into the clipboard as both a metafile and a bitmap. This allows applications that support both metafiles and bitmaps to retrieve the picture as a metafile if it needs to modify the picture or as a bitmap if it only needs to display the picture.

IClipboard predefines several system clipboard formats. In addition, any application can create and register additional private formats.

Before you can write any data to, or read any data from, the clipboard, you must first open it. Only a single application at a time can open the clipboard. If an application tries to open the clipboard but another application already has it open, it waits until the clipboard is available. The default behavior of the clipboard classes minimizes the time the clipboard is open.

If you use the default behavior of IClipboard, the clipboard functions that require an open clipboard will open it when needed and close it when finished. You turn off the default behavior of IClipboard when you explicitly open the clipboard by calling IClipboard::open. If you open the clipboard in this manner, functions in IClipboard will not close the clipboard when complete. If you explicitly open the clipboard, you must close the clipboard by calling IClipboard::close. You can turn off the default behavior of IClipboard to place different formats of your data on the clipboard without opening and closing it to write each format.

All clipboard operations must be associated with a window. You provide this window on the IClipboard constructor. If necessary, IClipboard makes this window the *owner* of the clipboard. The clipboard owner is the window responsible for the data put on the clipboard. It is also the window that the operating system sends messages to for events relating to the clipboard. The IClipboard object establishes this window as the system clipboard owner when you call IClipboard::empty. If you call IClipboard::owner before calling empty, your window will not be returned because it is not yet the system clipboard owner.

The clipboard classes support an advanced concept called *delayed rendering*. Delayed rendering allows you to wait until another application requests the data before you put the data on the clipboard. You activate delayed rendering by supplying 0 for the data when you call the clipboard functions to place data on the clipboard.

For more information on delayed rendering, see Chapter 36, “Supporting Direct Manipulation” on page 419.

You process clipboard events by creating and attaching an IClipboardHandler object to your clipboard owner window. In particular, if you use delayed rendering, you must attach an IClipboardHandler object to your clipboard’s window (the owner window). The window dispatcher calls this handler when a request is made to the clipboard for data that has not been placed there yet.

Because the clipboard should only be kept open for a short time, create IClipboard objects as temporary objects with a short lifetime. This helps ensure that the clipboard is only open for the time necessary.

The IClipboard destructor always closes the clipboard if it is still open.

Creating the Clipboard

You provide the clipboard owner on the IClipboard constructor and allow the functions needing an open clipboard to open the clipboard and close it when finished.

The following example uses an IClipboard object to copy text from an MLE into the clipboard and then to paste the data from the clipboard back into the MLE:

```
IBase::Boolean CommandHandler::command ( ICommandEvent&
event)
{
  switch(event.commandId())
  {
    case MI_COPY         :
    {
      IClipboard clipboard(event.window()->handle());
      clipboard.empty();
      clipboard.setText(edit.selectedText());
      return true;
    }

    case MI_PASTE          :
    {
      IClipboard clipboard(event.window()->handle());
      if (clipboard.hasText())
        edit.add(clipboard.text());
      return true;
    }
  }
  return false;
}
```

Note: An application can put only one item of one format into the clipboard. You can only put multiple items into the clipboard if each has a different format. Adding multiple items with the same format results in replacing the data.

Use the classes described below to create and manage a clipboard for your application:

IClipboard
Interface declaration class that creates a clipboard object.

IClipboard::Cursor
Nested class that iterates the available formats of data in the clipboard.

IClipboardHandler
Handler class to process the events that the clipboard sends to its owner. This includes requests to render clipboard data for formats that are put on the clipboard with delayed rendering.

📖 Refer to IClipboard in the *Open Class Library Reference* for more information about these classes.

To clear the contents of the clipboard use the IClipboard::ions:empty member function. This empties the contents of the clipboard and establishes the owner provided on open as the real clipboard owner. This function opens the clipboard if it is not already open and closes it after use unless the you have explicitly opened the clipboard by calling open.

Use the IClipboard::isOpen function to query the clipboard status. It returns true if the clipboard is open.

Clipboard Support

Moving Data Using the Clipboard

You can use the following IClipboard class member functions to move data to and from the clipboard:

setText Copies the passed text into shared memory and places it on the clipboard with the format IClipboard::textFormat.

setBitmap Copies the passed bitmap and places the handle on the clipboard with the format IClipboard::bitmapFormat.

setData Copies the passed data buffer and places it on the clipboard with the format specified. Register any private formats first by calling registerFormat. If “data” is 0, create an IClipboardHandler to process requests to render the data.

setHandle Places the passed handle on the clipboard.

hasText Returns true if the clipboard has data with the format IClipboard::textFormat.

hasBitmap Returns true if the clipboard has data with the format IClipboard::bitmapFormat.

hasData Returns true if the clipboard has data of any format.

text Returns data of the format IClipboard::textFormat as an IString object.

bitmap Copies data of the format IClipboard::bitmapFormat and returns an IBitmapHandle.

data Returns a `void*` value. This value can either be a pointer to the data being rendered or a handle depending on the format of the data.

This function always leaves the clipboard open. The caller must copy the data, if necessary, before closing the clipboard. Access to the data is lost after the clipboard is closed.

You can request delayed rendering of data by using the setData member function with the appropriate format and a 0 data pointer. Delayed rendering requires that you create an IClipboard handler to process requests to render the data when needed.

IClipboardHandler processes the clipboard event by creating an IEvent object and routing it to the appropriate virtual function. The virtual function allows you to supply your own specialized processing of the event. The return values from the virtual function specify whether the paint event is passed on to another handler object to be processed.

The dispatchHandlerEvent member function evaluates the event to determine if it is appropriate for this handler object to process. If it is, this function calls the virtual function used to process the event.

Clipboard Example

The following sample is taken from the CLIPBRD.HPP and CLIPBRD.CPP files found in the sample directory `\ibmcpp\samples\ioc\clipbrd`. This sample demonstrates how you can add clipboard support to a control, such as a container, that does not have built-in clipboard support. It also demonstrates how to use delayed rendering to put the data on the clipboard only when a user requests it during a paste operation.

In the .HPP file we do the following:

1. Declare a Department container object with an interface for rendering the object into a predefined format string and for initializing an existing object from a format string. This enables us to save the state of a Department object on a copy or cut operation so that we can create new Department objects during a paste operation.
2. Declare a ContainerCutPasteHandler object that is responsible for:
   - Creating and displaying a popup menu containing cut, copy, and paste choices.
   - Responding to user commands to cut, copy, and paste the content of Department objects on the clipboard.
   - Respond to clipboard handler requests to render the data to the clipboard.

```
⋮
#include <icliphdr.hpp>
#include <icmdhdr.hpp>
#include <istring.hpp>
#include <icnr.hpp>


//****************************************************************************
// Class:   Department                                                       *
//                                                                           *
// Purpose: Defines the data stored in the container for a Department.      *
//                                                                           *
//****************************************************************************
class Department : public IContainerObject {
public:
  Department ( const IString& name=IString(),
               const IString& address=IString())
     : IContainerObject (name),
       strAddress(address) {}

// Add functions to query and set the data.
virtual IString
 name       ( ) const,
 address    ( ) const;

virtual Department
 &setName    ( const IString& name),
 &setAddress ( const IString& address);

// Define the functions to render an object as both a
// private format and a normal text string, and to
// reconstruct the object from the private format.
IString
 asString        ( ) const,
 text            ( ) const;
Department
 &initializeFromString  ( const IString& renderedString);

// Define the separator character (a tilde) that separates
// the fields of the object in its string format.
static const IString
 separator,
 renderedFormat;

// Define a function to return the offset of the Address field.
static unsigned long
 offsetOfAddress ( ) { return offsetof(Department, strAddress); }

⋮

//**************************************************************************
// Class:   ContainerCutPasteHandler                                       *
//                                                                         *
// Purpose: Adds Clipboard support to the container for a Department       *
//          object.  This includes:                                        *
//          1) A container menu handler to show a pop-up menu with         *
//             cut, copy, and paste choices.                               *
//          2) A command handler to process the cut, copy, and paste       *
//             requests.                                                   *
//          3) A clipboard handler to process requests from the clipboard  *
//             to render data not yet placed on the clipboard.             *
//**************************************************************************
class ContainerCutPasteHandler : public ICommandHandler,
                                 public ICnrMenuHandler,
                                 public IClipboardHandler {
public:
ContainerCutPasteHandler (IContainerControl& container);

IContainerControl
 &container  ( ) { return cnr; }

protected:
// Define the command handler callback.
virtual Boolean
  command ( ICommandEvent& event);

// Define the pop-up menu callback.
Boolean
  makePopUpMenu(IMenuEvent& cnEvt);

// Define the callbacks to render data on the
// clipboard.
virtual Boolean
  clipboardEmptied     ( IEvent&        event ),
  renderFormat         ( IEvent&        event,
                         const IString& format),
109   renderAllFormats     ( IEvent&        event);
110
111 // Define a string object to use as a separator between fields
112 // for the private format.
113 static const IString
114  separator;
⋮
```

To support copying our Department object to the clipboard, we have devised a design to render a Department object as a data string and to create and initialize a new Developer object from data stored in such a data string. This data string contains fields separated by a character that we know does not exist in the data of our Department object. We define this separator character on line 57 and initialize it to a tilde character later in CLIPBRD.CPP on line 37.

The ContainerCutPasteHandler utilizes a very similar design to store a series of container objects on the clipboard. The ContainerCutPasteHandler must use a different separator to distinguish the parts of its data. In CLIPBRD.CPP on line 38, we define the caret (^) character as the separator between objects on the clipboard. When our handler receives a request to render the data to the clipboard in our private format, it creates a string with the following layout:

countn	separator (^)	object1	separator (^)	objectn

The number of objects in the string is stored as a text number in the first separator delimited field of the string. Also in the string, we put Department *object1* through *objectn* in their own separator delimited format with the following layout:

Department name	separator (˜)	Department address

For example, if the name field of a Department object is "Accounting" and the address field is "Building 4000," then the format string using the tilde character (˜) as the separator is:

```
"Accounting˜Building 4000"
```

If in addition to the Accounting department object above, we stored a "Sales" Department object with an address of "Building 5000," they would collectively appear on the clipboard as the string:

```
"2^Accounting˜Building 4000^Sales˜Building 5000"
```

Rather than copying our objects to the clipboard during the cut or copy operation, we have added support for delayed rendering. The design entails maintaining a collection of the objects cut or copied to the clipboard. When a user requests the data on the clipboard, our ClipboardCutPasteHandler's *renderFormat* routine is called to put the data on the clipboard. It iterates the objects in the collection and writes their data to the clipboard in the string format previously described.

A user that copies data to the clipboard and later pastes it into an application usually expects that the data will be the same as it was when it was first copied. To ensure this, any time the data of one of the objects in our collection changes or the object is removed from the container, we must force the delayed rendering mechanism to put the objects on the clipboard first. This support does not exist in our current clipboard sample.

This demonstrates that while delayed rendering has the potential for improving the performance of your application, it also increases its complexity. We therefore recommend that you first determine that you need to improve performance before deciding to add delayed rendering support to your application.

In the .CPP file we do the following:

1. Define our static Separator objects on line 36 and line 37.
2. Define the private format of our Department object on line 40.
3. Construct the ContainerCutPasteHandler on line 95, enable the event handlers on lines 100-102, and register the private clipboard format on line 105.
4. Process the clipboard operations cut, copy, and paste from line 115 to line 237.
5. Create and display the popup menu from line 245 to line 265.
6. Iterate our collection of clipboard objects and put the data onto the clipboard during the processing of the renderFormat function on line 306.
7. Initialize Department objects from our string format in the function *initializeFromString* on line 415 and build the string format for an object in the function *asString* on line 389.

```
⋮
 36 const IString Department::separator("~");
 37 const IString ContainerCutPasteHandler::separator("^");
 38
 39 // Define the private format of our Department object.
 40 const IString Department::renderedFormat("Department_rendered");
 89 /*-----------------------------------------------------------------------------
 90 | ContainerCutPasteHandler::ContainerCutPasteHandler                          |
 91 |                                                                             |
 92 | Construct the handlers, register our private clipboard format, and          |
 93 | attach the handlers to the container.                                       |
 94 -----------------------------------------------------------------------------*/
 95 ContainerCutPasteHandler :: ContainerCutPasteHandler ( IContainerControl& container)
 96       : cnr(container),
 97         objectList(new ICnrObjectSet())
 98 {
 99   // Enable the command, menu, and clipboard handlers.
100   ICommandHandler::handleEventsFor(&container);
```

```
  ICnrMenuHandler::handleEventsFor(&container);
  IClipboardHandler::handleEventsFor(&container);

  // Register the Department object's private format.
  IClipboard::registerFormat( Department::renderedFormat);
}


/*------------------------------------------------------------------------------
| ContainerCutPasteHandler::command                                            |
|                                                                              |
| Handle the command events associated with the clipboard (Cut, Copy,         |
| and Paste).                                                                  |
------------------------------------------------------------------------------*/
IBase::Boolean ContainerCutPasteHandler::command ( ICommandEvent& event)
{
  switch(event.commandId())
  {
    case MI_CUT           :
    case MI_COPY          :
    {
      // Empty the clipboard to establish ownership
      IClipboard clipboard(event.window()->handle());
      clipboard.empty();

      // Find the cursored object in the container.
      Department* cursoredObject = (Department*)(container().cursoredObject());

      // Utilize delayed rendering to put the data of the private
      // format on the clipboard.  Maintain an "objectList" collection
      // to keep track of the objects cut or copied onto the clipboard.
      // Then, use the collection to render objects in the private
      // format.  Store text data for the objects so that applications
      // that don't support the private format can paste the data into a
      // text editor.

      // Clear the collection used to keep track of clipboard objects.
      objectList->removeAll();

      // If the cursored object is selected, loop through all other
      // selected objects and store the objects in our set.
      if (container().isSelected(cursoredObject))
      {
        unsigned long count = 0;
        IString objectsAsText("");
        IContainerControl::ObjectCursor cursor(container(), IContainerObject::selected);
        for (cursor.setToFirst(); cursor.isValid(); cursor.setToNext())
        {
          count++;
          Department* selectedObject = (Department*)(container().objectAt(cursor));
          objectList->add(selectedObject);
          objectsAsText = objectsAsText +  selectedObject->text();
        }

        // Put the data on the clipboard.  We put our private
        // format first since it has the most information.
        // We use 0 for the data pointer of our private format
        // because we want to delay the rendering until
        // the paste operation.

        clipboard.setData(Department::renderedFormat, 0, 0);
        clipboard.setText(objectsAsText);

        // If this is a Cut, put the data on the clipboard
        // instead of using delayed rendering because we delete
        // the object. Therefore we request the data
        // to force the delayed rendering by our handler.
        if (event.commandId() == MI_CUT)
        {
          char* data = (char*)clipboard.data(Department::renderedFormat);
          container().deleteSelectedObjects();
        }
      }
      else
      {
        // If the object is not selected, repeat the above procedure
        // for the cursored object.
        objectList->add(cursoredObject);

        // Put the data on the clipboard (requesting
        // delayed rendering).
        clipboard.setData(Department::renderedFormat, 0,0);

        // If this is a Cut request, force the ClipboardCutPasteHandler
        // to render the data by requesting the data.  Then, delete the
        // cursored object.
        if (event.commandId() == MI_CUT) {
          char* data = (char*)clipboard.data(Department::renderedFormat);
          container().removeObject(cursoredObject);
          delete cursoredObject;
        }
      }
      return true;
    }
    case MI_PASTE         :
    {
      IClipboard clipboard(event.window()->handle());

      // If the clipboard has data of the private format,
      // get the data and build objects from it.
      // Note:  To see the text format of the data, paste
      // from the clipboard using any text editor that
      // supports the clipboard.
      if (clipboard.hasData(Department::renderedFormat))
      {
         IString strCount, strObject, objectsAsString;

         // Query the data on the clipboard.
         char* data = (char*)clipboard.data(Department::renderedFormat);

         // Parse the string into our data fields.
         data >> strCount >> separator >> objectsAsString;

         // Extract the number of objects stored in the string.
         unsigned long count = strCount.asUnsigned();

         // Turn refresh off to eliminate multiple painting.
         container().setRefreshOff();

         // Construct new objects from the data.
         for( int i=0; i<count; i++)
         {
           objectsAsString >> strObject >> separator >> objectsAsString;
           Department* department = new Department();
           department->initializeFromString(strObject);
           container().addObject(department);
         }

         // Enable refresh and refresh the container.
         container().setRefreshOn();
         container().refresh();
      }
      return true;
    }
  }
return false;
}


/*------------------------------------------------------------------------------
| ContainerCutPasteHandler::makePopUpMenu                                      |
|                                                                              |
| Create a pop-up menu with the clipboard actions.                             |
------------------------------------------------------------------------------*/
IBase::Boolean ContainerCutPasteHandler::makePopUpMenu(IMenuEvent& event)
{
  IPopUpMenu* popUpMenu = new IPopUpMenu(CNR_POPUPMENU,
                                         event.window());

  // Enable the allowable menu items in the pop-up menu.
  if (popupMenuObject()) {
    ((IContainerControl*)event.window())->setCursor(popupMenuObject());
    popUpMenu->disableItem(MI_PASTE);
  }
  else
  {
    popUpMenu->disableItem(MI_CUT);
    popUpMenu->disableItem(MI_COPY);
  }

  //  Show the menu.
  popUpMenu->setAutoDeleteObject();
  popUpMenu->show(event.mousePosition());
  return true;
}

/*------------------------------------------------------------------------------
| ContainerCutPasteHandler::clipboardEmptied                                   |
|                                                                              |
| This function is responsible for cleaning up data when the clipboard is      |
| emptied and has nothing to do for now.                                       |
------------------------------------------------------------------------------*/
IBase::Boolean ContainerCutPasteHandler::clipboardEmptied ( IEvent& )
{
  return true;
}

/*-------------------------------------------------------------------------------
| ContainerCutPasteHandler::renderFormat                                         |
|                                                                                |
| Put our private format data on the clipboard.                                  |
-------------------------------------------------------------------------------*/
IBase::Boolean ContainerCutPasteHandler::renderFormat( IEvent& event,
                                                const IString&  format)
{
  // Use the handler's collection to find the Department objects
  // whose data we need to put on the clipboard.
  // Note:  If data gets copied to the clipboard for a Department
  //        object and the object is deleted before the paste
  //        operation, we will not be able to supply the object's
  //        data.  If this is a problem, do not use delayed
  //        rendering or post a message to the container telling
  //        it to renderAllFormats whenever objects are deleted in
  //        the container.
  IClipboard clipboard(event.window()->handle());
  if (clipboard.hasData(Department::renderedFormat))
  {
     // Cursor the objects and build the string.
     ICnrObjectSet::Cursor cursor(*objectList);

     IString objectsAsString("");
     unsigned long count = 0;

     // Loop through our collection and query the object for
     // its data.
     for( cursor.setToFirst(); cursor.isValid(); cursor.setToNext())
     {
       count++;
       Department* department = (Department*)objectList->elementAt(cursor);

       // Document that no support is present for rendering objects
       // after they are removed from the container.
       IASSERTSTATE(container().containsObject(department));

       objectsAsString = objectsAsString + separator
                                       + department->asString();
     }
     // Build the final string and put the data on the clipboard.
     objectsAsString  =  IString(count) + objectsAsString;
     clipboard.setData(Department::renderedFormat,
                       (const char*)objectsAsString,
                       objectsAsString.size()+1);
  }
return true;

}


/*-------------------------------------------------------------------------------
| ContainerCutPasteHandler::renderAllFormats                                     |
|                                                                                |
| Pass this on to our function to render a single format since we only use      |
| one format with delayed rendering.                                          |
-----------------------------------------------------------------------------*/
IBase::Boolean ContainerCutPasteHandler::renderAllFormats( IEvent& event)
{
  return renderFormat(event, Department::renderedFormat);
}
⋮

/*-----------------------------------------------------------------------------
| Department::asString                                                        |
|                                                                             |
| Render the Department object as a String that we can later use to           |
| reconstruct a Department object.                                            |
-----------------------------------------------------------------------------*/
IString Department::asString ( ) const
{
  IString strObject = name() + separator + address();
  return strObject;
}

/*-----------------------------------------------------------------------------
| Department::text                                                            |
|                                                                             |
| Render the Department object as a text string with Name formatted to        |
| 30 characters, and address formatted to 50 characters with both             |
| followed by a new line character.  Use this format for storing a plain      |
| text format of our object on the clipboard.                                 |
-----------------------------------------------------------------------------*/
IString Department::text ( ) const
{
  IString strObject(name().subString(1,30) + address().subString(1,50) + "\n");
  return strObject;
}


/*-----------------------------------------------------------------------------
| Department::initializeFromString                                            |
|                                                                             |
| Set the fields of the object by parsing the passed string.                  |
-----------------------------------------------------------------------------*/
Department& Department::initializeFromString ( const IString& renderedString )
{
  IString strName, strAddress;
  renderedString >> strName >> separator >> strAddress;
  setName (strName);
  setAddress(strAddress);

  return *this;
}
```

Clipboard Support

41 Adding Tool Bars

A tool bar is a window whose buttons represent tools or menu items and actions. The tool bar can be located along the top, bottom, or sides of a frame window or can “float” and be positioned anywhere on your desktop. The floating tool bar can then be moved independently or you can “pin” the tool bar to the frame window. You can also have multiple tool bars with a variety of different tool bar buttons using text, bit maps, or both.

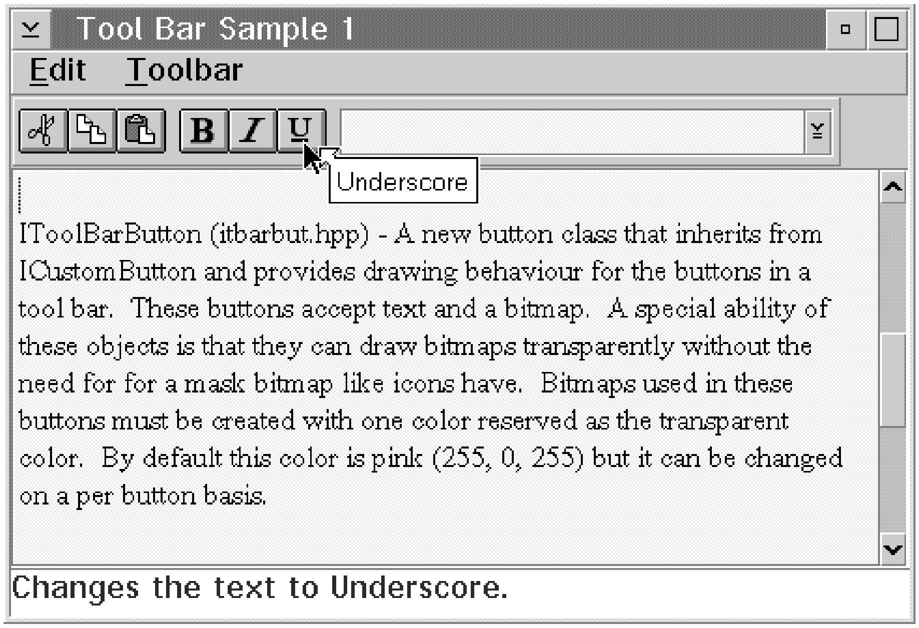
A new button class that inherits from ICustomButton and provides drawing behaviour for the buttons in a tool bar. These buttons accept text and a bitmap and can draw bitmaps transparently without the need for a mask bitmap. Bitmaps used in these buttons must be created with one color reserved as the transparent color. By default this color is pink (255, 0, 255) but it can be changed on a per button basis.

The classes that comprise the tool bar are also shipped with common text and bitmaps for frequently used functions found on a tool bar. This ensures a common look among applications and products. In addition, tool bar buttons support a standard style to ensure a consistant look across applications.

The tool bar is essentially a frame extension with buttons that represent common actions. You can add both fly over help and drag and drop support on the tool bar. You can use the direct manipulation support to rearrange the tool bar buttons within an application or add new buttons from a menu.

See “Adding Fly Over Help” on page 588 for more information on fly over help. See “Using Defaults for Tool Bars” on page 423 for more information on using direct manipulation with tool bars.

Figure 63 shows an example of a tool bar.



IToolBarButton (itbarbut.hpp) - A new button class that inherits from ICustomButton and provides drawing behaviour for the buttons in a tool bar. These buttons accept text and a bitmap. A special ability of these objects is that they can draw bitmaps transparently without the need for for a mask bitmap like icons have. Bitmaps used in these buttons must be created with one color reserved as the transparent color. By default this color is pink (255, 0, 255) but it can be changed on a per button basis.

Changes the text to Underscore.

Figure 63. Tool Bar Example

This figure is created using the sample code found in the `ibmcpp\samples\ioc\tbar1` directory.

Creating a Tool Bar

The IToolBar class creates and manages a tool bar area for a frame window. You can use the following classes to create tool bars:

IToolBar

Use objects of this class to create a tool bar, position that tool bar as a frame extension or as a floating tool bar, and add and remove tool bar buttons.

When you construct controls for a tool bar, you must explicitly add them to the tool bar using one of the following member functions:

- IToolBar::addAsFirst
- IToolBar::addAsLast
- IToolBar::addAsPrevious
- IToolBar::addAsNext

IToolBarButton

Creates and manages the tool bar button window. IToolBarButton provides support for displaying a bitmap, text, or bitmap and text on a tool bar button. If you specify a particular style for the view of your tool bar button, this style is changed when the button is added to the tool bar if it differs from the tool bar style itself.

The following classes are used by IToolBar and IToolBarButton to implement a tool bar:

IToolBarContainer
A set canvas that provides layout control for multiple tool bars within a frame extension. Objects of this class are created automatically by IToolBar objects to house one or more tool bars.

IToolBarTitle
Creates title bars that are smaller in height than the system title bar. This is created automatically whenever you create a floating tool bar using the IToolBar class.

IToolBarFrameWindow
Class representing a frame window to contain floating tool bars. IToolBarFrameWindows are created when you construct a floating tool bar.

IToolBarFrameWindows are different from other frame windows. The title bar height of IToolBarFrameWindows are smaller than other frame windows, with two buttons that appear where the frame window's minimize and maximize buttons normally are.

The left button is a toggle button that allows you to pin or attach a tool bar frame window to its owning frame window. This keeps the tool bar in the same location relative to its owner frame window when the owner is moved.

The right button allows you to expand and collapse the tool bar contained in the tool bar frame window. You can then reduce the size of the tool bar when it is not in use. Select the button again to expand the tool bar and show all the tool bar controls.

ICustomButton
Provide the base class for IToolBarButtons and create and manage the custom button window. Custom buttons allow the application to customize the button appearance by providing an ICustomButtonHandler to draw the button.

ICustomButtonDrawEvent
Provides information about a custom button event for ICustomButtonHandler event handling functions.

ICustomButtonDrawHandler
Processess ICustomButton drawing events. Attach objects of this class only to ICustomButton objects.

You can construct objects from the IToolBar class in the following ways:

1. The following IToolBar constructor creates a tool bar as the last tool bar in the tool bar area defined by the frameLocation. If *groupWithPreceding* is true, the tool bar is placed on the same row or column as the previous tool bar, if one exists. If an IToolBarContainer is needed for the location indicated, it is created when you use this contructor.

```
IToolBar ( unsigned long         identifier,
           IFrameWindow*         owner,
           Location              location = aboveClient,
           Boolean               groupWithPreceding = false,
           const Style&          style=defaultStyle());
```

2. Alternatively, you can construct a tool bar relative to an existing tool bar created with the first constructor (or with this constructor). This constructor adds the tool bar to the IToolBarContainer created when the *precedingToolBar* was created.

```
IToolBar ( unsigned long         identifier,
           IToolBar*             precedingToolBar,
           Boolean               groupWithPreceding = false,
           const Style&          style=defaultStyle());
```

Customizing Your Tool Bar

When you create an object from the IToolBarButton class, you can use a standard format to ensure that all your buttons have the same common appearance. Standard formatting controls the amount of area occupied by the bitmap (when visible) and the amount of area occupied by text (when visible). Standard formatting affects all of the tool bar buttons that have a style of IToolBarButton::standardFormat. The User Interface Class Library takes advantage of the standard formatting of tool bar buttons when painting the button. This therefore improves the performance.

The nested classes IToolBar::Style and IToolBarButton::Style provide a set of valid styles you can use upon construction for objects of the class.

For a complete listing of the styles and nested styles you can use with these classes, refer to the IToolBar classes in the *Open Class Library Reference*.

Tool Bar Example

The following example shows a customized tool bar that implements fly over help, direct manipulation support, and floating tool bars. This sample is found in the `ibmcpp\samples\ioc\tbar2` directory.

First, we do the following in the .HPP:

1. Define the main window for our MLE on lines 92 through 144.
2. Define our tool bars on lines 110 through 113.
3. Define the buttons we want to place on our tool bar on lines 126 through 134.
4. We also define our tool bar settings notebook on lines 149 through 158.

```
⋮
 85  //------------------------------------------------------------------
 86  // Editor
 87  //
 88  // This class is the main window of the sample problem.  It is
 89  // responsible for creating and managing all of the windows that
 90  // are used.
 91  //------------------------------------------------------------------
 92  class Editor : public IFrameWindow
 93  {
 94  public:
 95    Editor();
 96
 97  EditorMLE
 98   &editorWindow ( ) { return editWindow; }
 99
100  IFont
101   &editorFont ( ) { return editFont; }
102
103  Editor
104   &updateFontToolBar ( );
105
106  IToolBar
107   &toolBar ( unsigned long id );
108
109  private:
110  IToolBar
111    fileToolBar,
112    editToolBar,
113    fontToolBar;
114  IFlyText
115    flyText;
116  IStaticText
117    infoText;
118  IFlyOverHelpHandler
119    flyHelpHandler;
120  EditorMLE
121    editWindow;
122  EditorCommandHandler
123    commandHandler;
124  FontSelectHandler
125    fontSelectHandler;
126  IToolBarButton
127    openButton,
128    saveButton,
129    cutButton,
130    copyButton,
131    pasteButton,
132    boldButton,
133    italicButton,
134    underscoreButton;
135  IComboBox
136    fontCombo;
137  IMenuBar
138    menu;
139  IFont
140    editFont;
141  IWindow
142    *fileSubmenu,
143    *editSubmenu;
144  };
145
146  //----------------------------------------------------------------------
147  // ToolBarNotebook
148  //----------------------------------------------------------------------
149  class ToolBarNotebook : public IFrameWindow
150  {
151  public:
152    ToolBarNotebook ( Editor&   editor );
153  private:
154  Editor
155    &editorFrame;
156  INotebook
157    notebook;
158  };
⋮
```

In the .CPP file we then do the following:

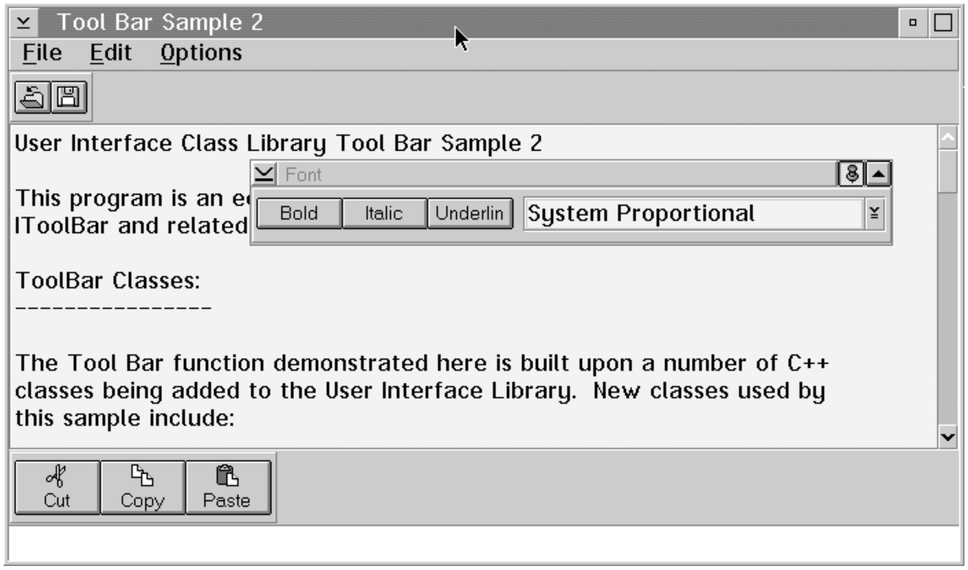
1. Construct and show the editor (MLE) main window on lines 37 through 40.
2. Construct tool bars within the main window constructor on lines 56 through 58.
3. Construct the fly over help short and long text and help handler on lines 59 through 61.
4. Add our tool bar buttons to the tool bars on lines 85 through 98.
5. Add the tool bar titles for floating tool bars on lines 118 through 120.
6. Add handlers for events, including fly over help on lines 132 through 145.
7. Enable direct manipulation from a menu choice on lines 148 through 151.

```
⋮
 29  //----------------------------------------------------------------------
 30  // Main routine
 31  //
 32  // This routine creates and shows the editor window
 33  //----------------------------------------------------------------------
 34  int main ()
 35  {
 36
 37     Editor editor;
 38     editor.show();
 39     editor.setFocus();
 40     IApplication::current().run();
```

```
   return 0;
}

//-------------------------------------------------------------------
// Editor::Editor
//
// This constructor initializes all of the windows (including the
// tool bar and tool bar buttons).
//
// Note: The tool bar buttons are created with standard identifiers
// that are provided by the class library (defined in icconst.h) for
// standard operations.
//-------------------------------------------------------------------
Editor::Editor ()
       :IFrameWindow("Tool Bar Sample 2"),
        fileToolBar(ID_FILE, this),
        editToolBar(ID_EDIT, &fileToolBar, true),
        fontToolBar(ID_FONT, &editToolBar, true),
        flyText(ID_FLYTEXT, this),
        infoText(ID_INFOTEXT, this, this),
        flyHelpHandler(&flyText, &infoText, 0),
        editWindow(ID_EDITOR, *this),
        commandHandler(*this),
        fontSelectHandler(*this),
        openButton      (IC_ID_OPEN,       &fileToolBar, &fileToolBar),
        saveButton      (IC_ID_SAVE,       &fileToolBar, &fileToolBar),
        cutButton       (IC_ID_CUT,        &editToolBar, &editToolBar),
        copyButton      (IC_ID_COPY,       &editToolBar, &editToolBar),
        pasteButton     (IC_ID_PASTE,      &editToolBar, &editToolBar),
        boldButton      (IC_ID_BOLD,       &fontToolBar, &fontToolBar,
                         IRectangle(), IToolBarButton::defaultStyle() |
                         IToolBarButton::noDragDelete ),
        italicButton    (IC_ID_ITALIC,     &fontToolBar, &fontToolBar,
                         IRectangle(), IToolBarButton::defaultStyle() |
                         IToolBarButton::noDragDelete ),
        underscoreButton(IC_ID_UNDERSCORE, &fontToolBar, &fontToolBar,
                         IRectangle(), IToolBarButton::defaultStyle() |
                         IToolBarButton::noDragDelete ),
        fontCombo(ID_FONTCOMBO, &fontToolBar, &fontToolBar, IRectangle(),
                  IComboBox::classDefaultStyle & IComboBox::simpleType |
                  IComboBox::dropDownListType),
        menu(ID_MAIN_WINDOW,this),
        editFont()
{
   // Add buttons to the file toolbar
   fileToolBar.addAsLast(&openButton,true);
   fileToolBar.addAsLast(&saveButton);

   // Add buttons to the edit toolbar
   editToolBar.addAsLast(&cutButton,true);
   editToolBar.addAsLast(&copyButton);
   editToolBar.addAsLast(&pasteButton);

   // Add buttons to the font toolbar
   fontToolBar.addAsLast(&boldButton,true);
   fontToolBar.addAsLast(&italicButton);
   fontToolBar.addAsLast(&underscoreButton);
   fontToolBar.addAsLast(&fontCombo,true);
```

```

⋮

    // Set up titles for toolbars when floating
    fileToolBar.setFloatingTitle(ID_FILE);
    editToolBar.setFloatingTitle(ID_EDIT);
    fontToolBar.setFloatingTitle(ID_FONT);

    // Setup the editor
    setClient(&editWindow);
    editWindow.setFont(editFont);
    editWindow.importFromFile("toolbar2.not");
    editWindow.setTop(1);

    // Add the Info frame extension
    addExtension(&infoText, IFrameWindow::belowClient);

    // Set up and add the help handler
    flyHelpHandler.setLongStringTableOffset(OFFSET_INFOTEXT);
    flyHelpHandler.setDefaultText(" ");
    flyHelpHandler.handleEventsFor(&fileToolBar);
    flyHelpHandler.handleEventsFor(&editToolBar);
    flyHelpHandler.handleEventsFor(&fontToolBar);

    // Attach the Command Handler to frame and toolbar
    commandHandler.handleEventsFor(this);
    commandHandler.handleEventsFor(&fileToolBar);
    commandHandler.handleEventsFor(&editToolBar);
    commandHandler.handleEventsFor(&fontToolBar);

    // Add the handler to change the font
    fontSelectHandler.handleEventsFor(&fontCombo);

    // Set up drag from menu
    fileSubmenu = new IWindow( menu.menuItem(ID_FILE).submenuHandle() );
    IDMHandler::enableDragFrom( (ISubmenu*) fileSubmenu );
    editSubmenu = new IWindow( menu.menuItem(ID_EDIT).submenuHandle() );
    IDMHandler::enableDragFrom( (ISubmenu*) editSubmenu );

⋮
```

The resulting MLE and tool bars are displayed in Figure 64



User Interface Class Library Tool Bar Sample 2

This program is an e
IToolBar and related

ToolBar Classes:

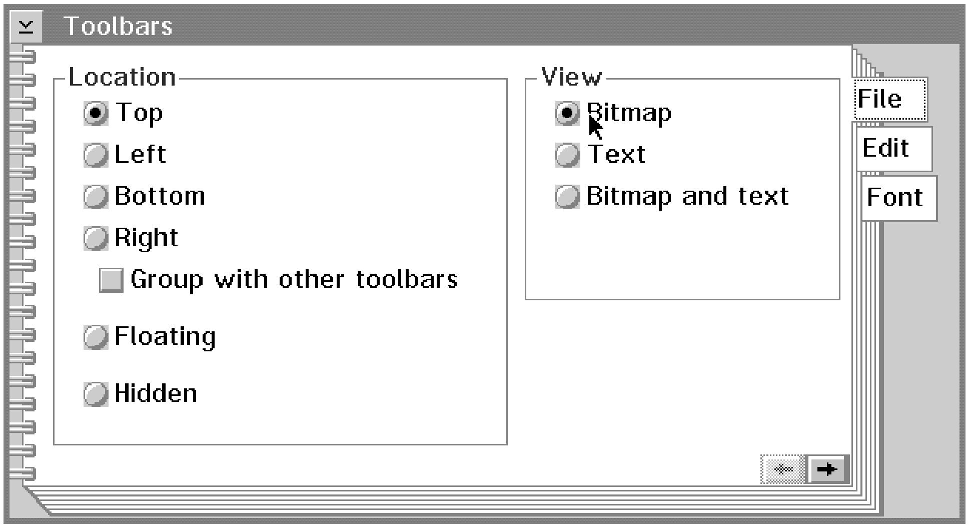
The Tool Bar function demonstrated here is built upon a number of C++
classes being added to the User Interface Library. New classes used by
this sample include:

Figure 64. Tool Bar Sample

You can use the settings notebook to chose where the tool bar will appear and what will appear on each tool bar button. In the preceding figure, there are three tool bars with different tool bar buttons. The floating tool bar, containing the font information, has been pinned to the owner window. The edit tool bar has been placed on the bottom of the window and contains both text and bitmaps on the buttons.

The settings notebook for the tool bars is displayed in Figure 65 on page 512

Tool Bars


Toolbars

Location
- ● Top
- ○ Left
- ○ Bottom
- ○ Right
  - □ Group with other toolbars
- ○ Floating
- ○ Hidden

View
- ● Bitmap
- ○ Text
- ○ Bitmap and text

Figure 65. Tool Bar Sample Settings Notebook

Several common bitmaps are provided for use when constructing IToolBarButton objects. The resource IDs for these bitmaps are defined in the ICCONST.H file. Figure 66 shows the commonly used bitmaps shipped with the User Interface Class Library.

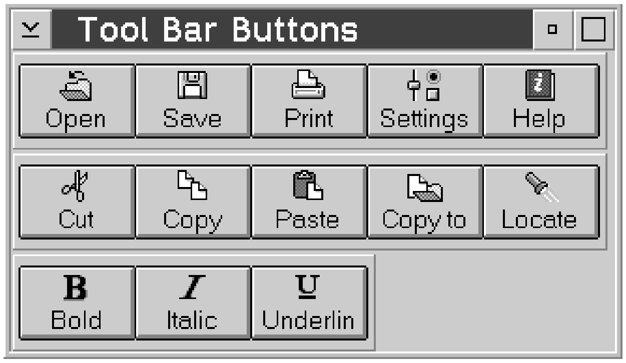


Figure 66. Tool Bar Bitmaps

42

Using Graphics in Your Application

The User Interface Class Library provides two-dimensional graphics support to fulfill your graphic element requirements for your current applications. In the future, we will provide Taligent graphics support as your portable graphics solution. The classes presented here will not be enhanced in future releases.

The graphics classes cover primitive graphical output mechanisms, such as lines, boxes, and curves; device control for controlling graphical output devices, such as the screen, printers, and plotters; attribute control such as colors, lines styles, fill patterns; font control; and use of bitmaps.

You can use these classes to customize the windows in your applications.

The graphics classes include the IGraphicContext class, IGraphic, and the graphic primitive classes. IGraphicContext wrappers a *presentation space* and provides much of the function in the OS/2 Presentation Manager Graphics Programming Interface (GPI) application program interfaces (APIs). A presentation space is a data structure, the equivalent of a blank piece of paper on which graphic images are created before being sent to an output device. An output device can be, for example, a printer or plotter, memory bitmap, or a display screen.

A *device context* is also a data structure. Its purpose is to translate graphics commands made to its associated presentation space into commands that the physical device can convert to displayed information.

Graphics primitives are building blocks provided by the programming interface from which you can construct two-dimensional pictures. The classes provided for constructing and drawing the graphic primitives include line, arc, pie, chord, text, polyline, polygon, ellipse, box, and bitmap.

Each graphic primitive has a set of properties that further defines its appearance. A line, for example, can be drawn in different colors and different widths; it can be dotted, solid, or even invisible. These properties are called *primitive attributes.* Primitive attributes are set for the type of primitive rather than for a single instance of the primitive.

All primitive attributes have default settings that apply when a presentation space is first created and continue to apply until you change them. When an attribute value is set, it remains in effect until you change it.

Each primitive can optionally contain an IGraphicBundle object. The IGraphicBundle class represents a collection of drawing attributes. Some of the attributes contained within a bundle may not apply to a specific primitive and will be ignored. For example, a line does not have fill color and that attribute, even if it is set, are ignored.

Each primitive can also contain an ITransformMatrix object. This object represents a transformation matrix that is concatenated to the existing model space transform before drawing the primitive.

The User Interface Class Library provides the following graphics classes for use in your applications:

- IGArc
- IGBitmap
- IGChord
- IGEllipse
- IGLine
- IGList
- IGList::Cursor
- IGPie
- IGPolygon
- IGPolyline
- IGRectangle
- IGRegion
- IGString
- IGraphic
- IGraphicBundle
- IGraphicContext
- IG3PointArc
- IPointArray
- ITransformMatrix

See “2D Graphic Classes” on page 662 for the class hierarchy for the User Interface Class Library graphic classes.

Adding Graphic Primitives to Your Applications

IGraphic is an abstract base class that provides common behavior for all of the two-dimensional graphic classes. All of the graphic primitives derive from this class and this common behavior includes bounding rectangle information, accessing bundle attributes, transforming a graphic object, detecting if a graphic object has been selected, and identifying a graphic object. You cannot instantiate an object of this class directly.

An IGraphic object can optionally contain an IGraphicBundle and an ITransformMatrix. You can set a graphic bundle onto a graphic object so that the attributes set in the bundle are used when you draw a graphic object.

If you use any of the world transform functions or set a transform matrix on a graphic object, then the transform matrix contained in the graphic object is used when you draw a graphic object.

A *transformation* is an operation performed on a graphic object that changes the object in one of the following ways:

- Translation
- Rotation
- Scaling
- Shearing

Transformations enable an application to control the location, orientation, size, and shape of graphic objects on an output device.

The transformation of graphic objects can be conceptually divided into a series of distinct transformations applied from 1 logical stopping point to another. *Coordinate spaces* are used as a method of conceptualizing these logical stopping points. The coordinate spaces are concepts used to explain and manipulate the transformation process.

The *world coordinate space* is where most drawing coordinates are specified. The world coordinate space is a grid that provides a reference scale for what is being drawn on the presentation space. The components of a picture defined in world coordinate space are often defined to a scale convenient to only that component. Applications also can define each component, or subpicture, starting at the origin (0,0). This enables applications to define the scale of a subpicture and the location of the subpicture separately.

After subpictures are defined in world coordinate space, they undergo a transformation before they appear on an output device. If an application has not specifically applied a transformation on a subpicture, by default the identity transform is applied. This makes no change to the subpicture.

See “Defining a Transformation Matrix” on page 528 and the *Open Class Library Reference* for more information.

Setting Attributes for Drawing Primitives

You can set drawing attributes for graphic objects (such as IGLine, IGArc, and IGPolyline) using the IGraphicBundle class. You can use this class to set an attribute for a graphic object that does not support the attribute.

Graphics

The graphic bundle class allows you to selectively change drawing attributes from the attributes currently set for a graphic context. If a graphic object does not contain a graphic bundle, the current graphic context drawing attributes are used when you draw a graphic object. If you call a set attribute function of a graphic bundle object and set the graphic bundle onto a graphic object, this graphic bundle's attributes override the drawing attributes for the graphic context when you draw that object. For drawing attributes that you have not changed using a set attribute function, the current drawing attributes set in the graphic context are used when you draw a graphic object.

Each drawing attribute has four functions associated with it:

set Sets a drawing attribute on the graphic bundle

query Obtains a drawing attribute set on a graphic bundle.

has Determines if a drawing attribute is set on a graphic object.

reset Uses the current graphic context drawing attribute instead of the drawing attribute set in the graphic bundle.

You can use the following IGraphicBundle member functions to change the drawing attributes of a graphic object:

background mix mode
determines how an existing drawing is combined with the background object of a graphic object.

color attributes
Controls the use of pen, fill, and background colors for a graphic bundle.

draw operation
Controls the methods used to draw any of the closed graphic objects.

mix mode
Controls the use of pen and fill colors with existing drawings.

pen and fill Patterns
Controls the pattern used when you draw lines that have a pen width greater than one or when you fill a closed figure.

pen type
Defines the way lines, arc, polylines, and the frame on closed figures are drawn.

pen width
Controls the width of the pen when you draw lines or the width of the frame when you draw a closed figure. A pen width greater than one is always a solid pen type.

pen ending style
Controls the shape of the unattached end of a line. The pen width must be greater than one for the style to have an affect.

pen joining style
Controls the shape formed by two intersecting lines. The pen width must be greater than one for the style to have an affect.

Figure 67 shows the different pen types.

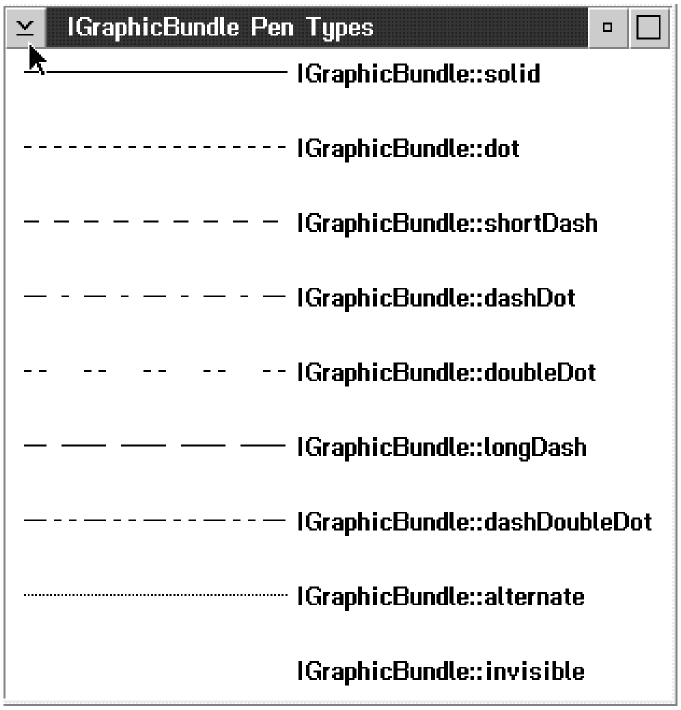

IGraphicBundle Pen Types
IGraphicBundle::solid
IGraphicBundle::dot
IGraphicBundle::shortDash
IGraphicBundle::dashDot
IGraphicBundle::doubleDot
IGraphicBundle::longDash
IGraphicBundle::dashDoubleDot
IGraphicBundle::alternate
IGraphicBundle::invisible

Figure 67. Pen Types

Figure 68 shows the different pen joining styles.

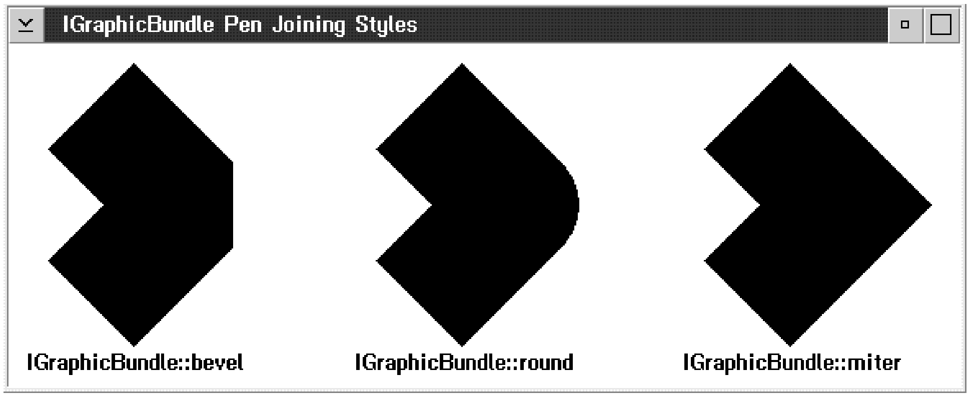


Figure 68. Pen Joining Styles

Figure 69 shows the different pen and fill patterns.

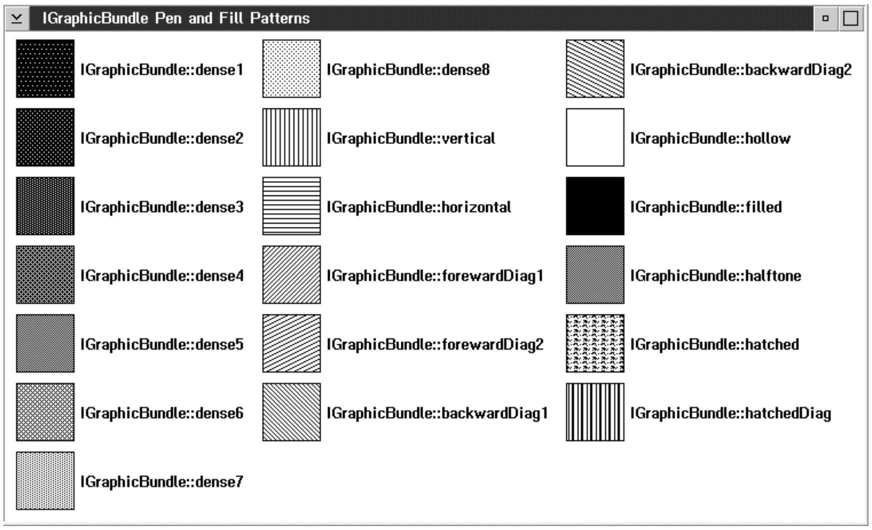


Figure 69. Mix Modes

Refer to the *Open Class Library Reference* for the complete listings of supported styles.

Drawing Lines and Arcs

Simple drawing applications use line and arc primitives as drawing tools or building blocks for more complex pictures, such as geometric objects, pie charts, and bar graphs. Both lines and arcs are governed by the following attributes:

- pen color

- mix mode
- pen width
- pen type
- pen ending style

Use the IGLine class to create two-dimensional line segments specifying the starting (IGLine::setStartingPoint) and ending points (IGLine::setEndingPoint) of the line segment.

The IGPolyline class allows you to create a series of line segments drawn starting from the first point and then connecting all remaining points. You use IGPolyline::setPoints to set the points used to define the polyline.

Using the IGArc class, you can create two-dimensional arcs by specifying a bounding rectangle, a start angle, and a sweep angle. The rectangle you specify in the constructor is the enclosing rectangle of an ellipse. The start angle and sweep angles specify an arc section of this ellipse.

IG3PointArc objects are two-dimensional arcs created by specifying three points that the arc passes through: the starting point, intermediate point, and ending point of the arc. The three points specify the arc of a circle unless you apply a transform to the object.

Displaying Areas, Polygons, and Regions

You can use the following classes to create two-dimensional closed figures:

IGPolygon

Creates a two-dimensional closed figure from a series of line segments. The series of line segments is drawn starting from the first point and connecting all remaining points. If the first and last points are not the same, this class draws a line from the last point to the first point to close the figure.

IGRectangle

Draws two-dimensional rectangles. An IGRectangle can be filled, framed, or filled and framed.

You can optionally round the corners of the rectangle by specifying the full length of the horizontal and vertical axes of an ellipse. The corners of the rectangle are rounded by a quarter of the ellipse.

IGEllipse

Draws two-dimensional ellipses. An IGEllipse can be filled, framed, or filled and framed.

Graphics

IGPie

Creates a two-dimensional pie slice of an ellipse. The IGPie can be filled, framed or filled and framed. The rectangle you specify in the constructor is the enclosing rectangle of an ellipse. The start angle and sweep angles specify a pie section of this ellipse.

IGChord

Creates a two-dimensional closed figure created from the chord of an ellipse. The IGChord can be filled, framed, or filled and framed. The rectangle you specify in the constructor is the enclosing rectangle of an ellipse. The start angle and sweep angles specify a chord section of this ellipse.

IGRegion

Creates graphic objects that can be composed of one or more closed figures. You can use an IGRegion to construct a shape from one or more closed figures. You can draw the region on a graphic context or use it as a clip region when drawing other graphic objects on a graphic context.

The coordinates you use to define a region are specified in device space. For this reason the world transform functions that are declared in IGraphic have no effect and are overridden as private functions in IGRegion.

For a listing of the graphic bundle attributes that affect these objects and how to construct objects of these classes, see the *Open Class Library Reference*

Using Character Strings

The IGString class is a graphic object that allows you to draw text. When you construct objects of this class you must provide a location point of where the text drawing starts. The text alignment and font direction determine where the text is positioned relative to that location. You can also associate a font with it that will be used when you draw the IGString object.

Objects of the IFont class manage the use of fonts. Use these objects to select a font through the IFont functions. You can also use the font dialog to get font information and set the font when drawing text.

The IFont class attempts to match the requested font but if it cannot find an exact match, IFont uses the nearest match. An IFont object represents a particular font that is available on the system. It does not represent a font actually being used by a control or the currently selected font for a presentation space.

Refer to the *Open Class Library Reference* for more information about the IFont classes

Working with Bitmaps

The IGBitmap class is used to create, copy, modify, and draw bitmaps. IGBitmap objects can be created from existing bitmap handles, from bitmap resources, from a rectangular area of a graphic context, or directly from an image file.

Once a bitmap has been created, you can save the bitmap in any of the supported image file formats.

If the bitmap is a color bitmap, none of the graphic bundle attributes affects the appearance of the bitmap. If the bitmap is a *monochrome* bitmap, one bit-per-plane, then the following graphic bundle attributes affect its appearance:

- Pen color
- Background color
- Mix mode
- Background mix mode

The following sample demonstrates how to use the IGBitmap class. This sample is located in the `ibmcpp\samples\ioc\2d-bmap` directory and shows how you can work with bitmap files.

First, in the .HPP we do the following:

1. Declare our drawing area paint handler on line 32.
2. Declare our command handler to process events on line 49 through 58.
3. Declare our drawing area on lines 60 through 95.
4. Declare a pointer to our IGBitmap object.
5. On lines 104 through 107 we declare our main window including our drawing area.
6. Load an image based on a string on line 115.

```
26 //*************************************************************************
27 // Class:   DrawingAreaPaintHandler                                       *
28 //                                                                        *
29 // Purpose: Draw the bitmap.                                              *
30 //                                                                        *
31 //*************************************************************************
32 class DrawingAreaPaintHandler : public IPaintHandler
33 {
34 typedef IPaintHandler
35   Inherited;
36
37 protected:
38
39 virtual Boolean
40   paintWindow( IPaintEvent& event );
41 };
42
```

```
//**************************************************************************
// Class:   MainCommandHandler                                             *
//                                                                         *
// Purpose: Handle command events for the bitmap sample program.           *
//                                                                         *
//**************************************************************************
class MainCommandHandler : public ICommandHandler
{
typedef ICommandHandler
  Inherited;

protected:

virtual Boolean
  command( ICommandEvent& event );
};

class DrawingArea : public IDrawingCanvas
{
typedef IDrawingCanvas
  Inherited;

public:

  DrawingArea( unsigned long id, IWindow* parent, IWindow* owner );
virtual
   DrawingArea();

DrawingArea
  &loadBitmap( const IString& imageFile );

IGBitmap*
  bitmap() const;

DrawingArea
  &setClipStyle ( unsigned long style ) { fStyle = style; return *this; }
unsigned long
  clipStyle( ) const { return fStyle; }

protected:

virtual ISize
  calcMinimumSize     ( ) const;

private:

DrawingAreaPaintHandler
  drawingAreaPaintHandler;
IGBitmap*
  fBitmap;
unsigned long
  fStyle;
};

//**************************************************************************
// Class:   MainWindow                                                     *
//                                                                         *
// Purpose: Main Window for C++ Hello World sample application             *
//          It is a subclass of IFrameWindow                               *
//                                                                         *
//**************************************************************************
class MainWindow : public IFrameWindow
{
typedef IFrameWindow
  Inherited;

public:                                  //Define the Public Information
  MainWindow(unsigned long windowId);    //Constructor for this class
virtual
   MainWindow();

virtual MainWindow
  &loadImageFile( const IString& imageFile );

virtual MainWindow
  &modifyBitmap( unsigned long eventId );

DrawingArea
  &drawingArea( ) { return fDrawingArea; }

private:                                 //Define Private Information
  IViewPort          fViewPort;
  DrawingArea        fDrawingArea;
  MainCommandHandler  fMainCommandHandler;
};
```

In the .CPP file, we then do the following:

1. List the actions we can use to modify the bitmap on lines 85 through 110. For example, on line 99 we rotate the bitmap by 90 degrees.
2. Define our command handler on line 124.
3. Load a bitmap using a file dialog on lines 129 through 148.
4. List the actions we can perform on the bitmap calling the functions defined above.
5. Load the image file, calling our drawing area on line 187.
6. Define the handler for painting the window on line 238.
7. On lines 240 through 255, we define our graphic context and graphic region.
8. We clip the bitmap using an ellipse on lines 262 through 279 and a rectangle on lines 280 through 292.
9. On lines 297 through 307 we rotate the bitmap.
10. We then draw the bitmap on line 314 through 317.

```
⋮
 85  MainWindow& MainWindow::modifyBitmap( unsigned long eventId )
 86  {
 87    IGBitmap* bitmap(fDrawingArea.bitmap());
 88    if (bitmap)
 89    {
 90      switch (eventId)
 91      {
 92        case  IDM_REFLECTHORZ:
 93          bitmap->reflectHorizontally();
 94        break;
 95        case  IDM_REFLECTVERT:
 96          bitmap->reflectVertically();
 97        break;
 98        case  IDM_ROTATE90:
 99          bitmap->rotateBy90();
100        break;
```

```
      case  IDM_ROTATE180:
        bitmap->rotateBy180();
      break;
      case  IDM_ROTATE270:
        bitmap->rotateBy270();
      break;
      case  IDM_TRANSPOSE:
        bitmap->transposeXForY();
      break;
    } /* endswitch */
    ISize bitmapSize(bitmap->size());
    fDrawingArea.sizeTo(bitmapSize);
    fViewPort.setViewWindowSize(bitmapSize);
    fDrawingArea.refresh();
  }
  return *this;
}

/*------------------------------------------------------------------------------
| MainCommandHandler::command                                                 |
|                                                                             |
|                                                                             |
------------------------------------------------------------------------------*/
IBase::Boolean MainCommandHandler::command( ICommandEvent& event )
{
  Boolean fProcessed = false;
  switch (event.commandId())
  {
    case  IDM_FILEOPEN:
    {
      MainWindow *mainWindow((MainWindow*)event.window());

      IFileDialog::Settings fsettings;      //                                      .
      fsettings.setTitle("Load an image file");//Set Open Dialog Title from Resource  .
      fsettings.setFileName("*.bmp");          //Set FileNames to *.hlo               .

      IFileDialog fd(                          //Create File Open Dialiog             .
        IWindow::desktopWindow(),              //  Parent is Desktop                  .
        mainWindow, fsettings);                //  Owner is me with settings          .
      if (fd.pressedOK())                      //Check if ok from file open dialog    .
      {
        mainWindow->loadImageFile(fd.fileName());
        IMenuBar menuBar(mainWindow->id(), mainWindow);
        menuBar.setAutoDestroyWindow(false);

:
      }
    }
    break;
    case  IDM_QUIT:
    {
      IFrameWindow *frameWindow((IFrameWindow*)event.window());
      frameWindow->close();
    }
    break;
    case  IDM_REFLECTHORZ:
    case  IDM_REFLECTVERT:
    case  IDM_ROTATE90:
    case  IDM_ROTATE180:
    case  IDM_ROTATE270:
    case  IDM_TRANSPOSE:
    {
      MainWindow *mainWindow((MainWindow*)event.window());
      mainWindow->modifyBitmap( event.commandId() );
    }
    break;

    case  IDM_CLIPCIRCLES:
    case  IDM_CLIPSQUARES:
    case  IDM_CLIPRAD:
    case  IDM_CLIPNONE:
    {
      MainWindow *mainWindow((MainWindow*)event.window());
      mainWindow->drawingArea().setClipStyle( event.commandId() );
      mainWindow->drawingArea().refresh();
    }
    break;
  } /* endswitch */

  return fProcessed;
}

/*------------------------------------------------------------------------------
| MainWindow::loadImageFile                                                    |
|                                                                              |
|                                                                              |
------------------------------------------------------------------------------*/
MainWindow& MainWindow::loadImageFile( const IString& imageFile )
{
  fDrawingArea.loadBitmap( imageFile );
  fViewPort.setViewWindowSize( fDrawingArea.bitmap()->size());
  return *this;
}

DrawingArea::DrawingArea( unsigned long id, IWindow* parent, IWindow* owner )
  : DrawingArea::Inherited( id, parent, owner, IRectangle(),
                            IDrawingCanvas::defaultStyle() &  IDrawingCanvas::useDefaultPaintHandler ),
    fBitmap(0),
    fStyle(IDM_CLIPNONE)
{
  drawingAreaPaintHandler.handleEventsFor(this);
}

DrawingArea:: DrawingArea()
{
  drawingAreaPaintHandler.stopHandlingEventsFor(this);
  if (fBitmap)
    delete fBitmap;
}

IGBitmap* DrawingArea::bitmap( ) const
{
  return fBitmap;
}

DrawingArea& DrawingArea::loadBitmap( const IString& imageFile )
{
  if (fBitmap)
    delete fBitmap;
  fBitmap = new IGBitmap( imageFile );
  sizeTo(fBitmap->size());
  refresh();
  return *this;
}

ISize DrawingArea::calcMinimumSize( ) const
{
  if (fBitmap)
    return fBitmap->size();
  else
    return ISize();
}

```

```
/*------------------------------------------------------------------------------
| DrawingAreaPaintHandler::paintWindow                                         |
|                                                                              |
|                                                                              |
------------------------------------------------------------------------------*/
IBase::Boolean DrawingAreaPaintHandler::paintWindow( IPaintEvent& event )
{
  // Get a graphic context
  IGraphicContext gc(event.presSpaceHandle());

  // Get a pointer to the current bitmap if one exists.
  IGBitmap* bitmap(((DrawingArea*)event.window())->bitmap());

  // Get the dimensions of the window
  IRectangle windowRect(IPoint(),((DrawingArea*)event.window())->size());

  // paint the current background color
  event.clearBackground();

  // Query the current clipping style
  unsigned long clipStyle(((DrawingArea*)event.window())->clipStyle());

  IGRegion region;
  IGRegion oldClipRegion( gc.clipRegion() );

  // Clear the current clip region so that the oldClipRegion can
  // be used in region operations.
  gc.clearClipRegion();

  switch (clipStyle)
  {
    case IDM_CLIPCIRCLES:
    {
      IGRectangle rect( windowRect );
      region -= rect;
      IGEllipse ellipse( windowRect );
      region += ellipse;
      windowRect.shrinkBy( 25 );
      ellipse.setEnclosingRect( windowRect );
      region -= ellipse;
      windowRect.shrinkBy( 25 );
      ellipse.setEnclosingRect( windowRect );
      region += ellipse;

⋮
      region &= oldClipRegion;
    }
    break;
    case IDM_CLIPSQUARES:
    {
      IGRectangle rect( windowRect );
      region += rect;
      windowRect.shrinkBy( 25 );
      rect.setEnclosingRect( windowRect );
      region -= rect;
      windowRect.shrinkBy( 25 );
      rect.setEnclosingRect( windowRect );
      region += rect;

⋮
      region &= oldClipRegion;
      break;
    }
    case IDM_CLIPRAD:
    {
      IGPie pie( windowRect, 0, 60 );
      region += pie;
      pie.setStartAngle( 120 );
      region += pie;
      pie.setStartAngle( 240 );
      region += pie;
      region &= oldClipRegion;
      break;
    }
    case IDM_CLIPNONE:
      region = oldClipRegion;
    break;
  } /* endswitch */

  // set the clip region
  gc.setClipRegion( region );

  // draw the bitmap if we have one
  if (bitmap)
  {
    bitmap->drawOn(gc);
  }

  // clear the current clip region
  gc.clearClipRegion( );

  return true;
}
```

The resulting window is displayed in Figure 70

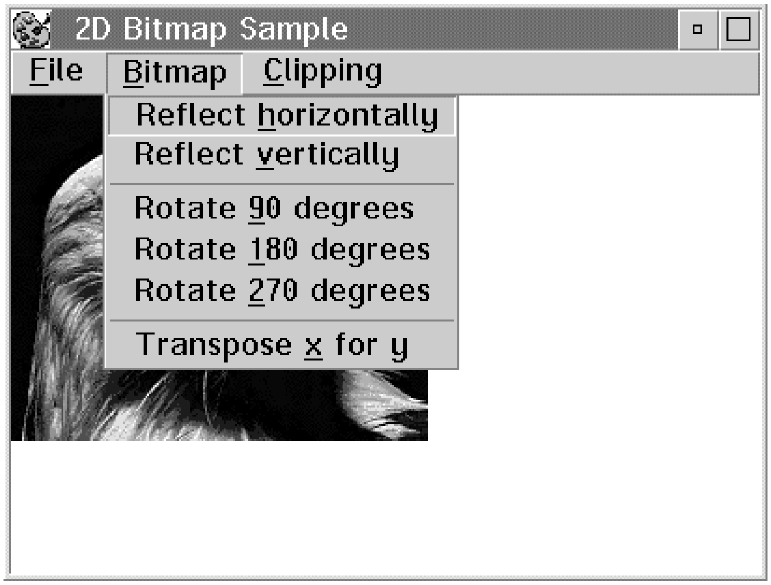


Figure 70. Bitmap Sample

Grouping Graphic Objects

The IGList class is an ordered collection of IGraphic objects. The IGraphic objects are arranged so that each IGList object has a first and a last IGraphic object, each IGraphic object except the last has a next IGraphic object, and each IGraphic object but the first has a previous IGraphic object.

An IGList allows you to group simple IGraphic objects to compose a complex picture. At any time you can add additional IGraphic objects to the IGList or remove any IGraphic objects from it. You can also add the same IGraphic object to the list multiple times to replicate part of a picture.

Because IGList inherits from IGraphic, you can add an IGList to an IGList. You can also use any of the transform functions inherited from IGraphic on an IGList. Transforms applied to an IGList affect all IGraphic objects in the IGList. You can easily construct complex pictures and transform them as a single entity.

When you draw an IGList, it iterates through the graphic objects contained in the list. IGList recursively calls the drawOn function for all nested IGLists. Also, the attributes you use to draw the graphic objects contained in the list are those of the graphic bundle applied to the IGList with one exception. When a graphic object contains a graphic bundle that has the same attributes set, the graphic object's bundle attributes override the IGList's graphic bundle attributes.

The member functions provided by the IGList class allow you to add graphics objects at varying positions within the list, remove objects, reorder the list, and query list information and retrieve the graphic objects in the list.

The IGList::Cursor class iterates through graphic objects contained in an IGList. The Cursor class has three constructors that control how to iterate through the IGList. The class iterates through top-level objects only. If you nest an IGList inside an IGList, the cursor does not iterate through the graphic objects contained within the nested IGList.

Refer to the *Open Class Library Reference* for more information about the IGList and IGList::Cursor classes.

Defining a Transformation Matrix

The ITransformMatrix class is used to represent a 3x3 transformation matrix.

You can use ITransformMatrix objects to quickly construct transformation matrixes for use with IGraphic::setTransformMatrix or with the native graphic programming interface.

If you use any of the world transform functions or set a transform matrix on a graphic object, then the transform matrix contained in the graphic object is used when you draw a graphic object.

📖 Refer to the *Open Class Library Reference* for more information.

Using the Drawing Functions

The IGraphicContext class renders graphic objects on a device. IGraphicContext contains the graphic state information used when drawing graphics. The graphic state includes the current attribute bundle, the current world space transform matrix, and the current clip region.

PM The IGraphicContext class wrappers the presentation space data types.

The IGraphicContext class also contains a static collection of default drawing attributes that initialize the attributes of the graphic context when you create it. You can change any of these default drawing attributes so that any graphic context created subsequently is initialized with these new drawing attributes.

You can also use other native graphic programming interfaces whose function is not included in the User Interface Class Library graphics classes by accessing either the presentation space handle or device context by using the handle member function as the first parameter to the graphic programming interface.

Adding Handlers to Graphical Objects

Use the ITrackingHandler class to handle events resulting from a user changing a control's input value without releasing the mouse, such as rotating the circular slider or moving the arm of a slider. ITrackingHandler objects process input tracking events for the ISlider and ICircularSlider controls.

You create a handler derived from ITrackingHandler and attach it to either the control whose input users can change or to the control's owner window. Call IHandler::handleEventsFor to pass the appropriate control window or owner window to the edit handler.

Two-dimensional Graphics Samples

The following sample demonstrates how to use the two-dimensional graphics classes. This sample is located in the `ibmcpp\samples\ioc\2d-draw`

First, in the .HPP file, we do the following:

1. Declare our drawing area on line 46.

2. Declare all our graphic objects.

3. Declare our main frame window on line 155.

```
⋮
39  //**************************************************************************
40  // Class:   DrawingArea                                                    *
41  //                                                                         *
42  // Purpose: Subclass of IDrawingCanvas.  Class contains the handlers       *
43  //          necessary for interactive drawing of the graphic objects.      *
44  //                                                                         *
45  //**************************************************************************
46  class DrawingArea : public IDrawingCanvas,
47                      public IMouseHandler
48  {
49  public:
50    DrawingArea  ( unsigned long windowId,
51                   IWindow* parent,
52                   IWindow* owner,
53                   const IRectangle& intial = IRectangle() );
54  virtual
55     DrawingArea ( );
56
57  enum DrawState {
58    drawing,
59    waitForInput,
60    notDrawing
61    };
62
63  DrawingArea
64    &setDrawState        ( const DrawState newState = drawing );
65  DrawState
66    drawState            ( ) const { return dState; }
67
68  DrawingArea
69    &setBitmapFileName   ( const IString& bitmapFile )
70                           { currentBitmap = bitmapFile;
71                             return *this; }
72  IString
73    bitmapFileName       ( ) const
74                           { return currentBitmap; }
75
76  enum DrawObject {
77    pointer = PALLET_NORM,
78    line,
79    freeHand,
80    rectangle,
81    ellipse,
82    polyline,
83    polygon,
84    arc,
85    pie,
86    chord,
87    text,
88    bitmap
89    };
90
91  virtual DrawingArea
92    &setDrawObject       ( const DrawObject drawObject )
93                           { currentObj = drawObject;
94                             return *this; }
95  virtual DrawObject
96    drawObject           ( ) const {return currentObj;}
97
98  virtual IGraphicBundle
```

```
  &graphicBundle       ( ) { return currentBundle; }

virtual DrawingArea
  &setCurrentFont      ( const IFont& font );
virtual IFont
  currentFont          ( ) const;

protected:

virtual Boolean
  mouseMoved           ( IMouseEvent&        event ),
  mouseClicked         ( IMouseClickEvent&   event ),
  mousePointerChange   ( IMousePointerEvent& event );

virtual DrawingArea
  &button1Down         ( const IPoint&       point ),
  &button1Up           ( const IPoint&       point ),
  &button1DoubleClick  ( const IPoint&       point ),
  &button2Down         ( const IPoint&       point ),
  &button2Up           ( const IPoint&       point );

private:
  IGraphicContext    gc;
  IFont              currentfont;
  IGraphicBundle     currentBundle;
  IString            currentBitmap;
  DrawState          dState;
  DrawObject         currentObj;
  IGraphic*          iGraphic;
  IGraphic*          moveGraphic;
  IGRectangle        moveRect;
  IPoint             startingPt;
  IPoint             previousPt;
  IPoint             tempPt;
  unsigned long      pointCount;
  IPointerHandle     ptrLine;
  IPointerHandle     ptrDraw;
  IPointerHandle     ptrRectangle;
  IPointerHandle     ptrEllipse;
  IPointerHandle     ptrPolyline;
  IPointerHandle     ptrPolygon;
  IPointerHandle     ptrArc;
  IPointerHandle     ptrPie;
  IPointerHandle     ptrChord;
  IPointerHandle     ptrText;
  IPointerHandle     ptrBitmap;
  IPointerHandle     ptrCurrent;
};

//**************************************************************************
// Class:   MainWindow                                                     *
//                                                                         *
// Purpose: Main Window for C++ 2D-Draw sample application                 *
//          It is a subclass of IFrameWindow                               *
//                                                                         *
//**************************************************************************
class MainWindow : public IFrameWindow,
                   public ICommandHandler,
                   public IMenuDrawItemHandler
{
public:                               //Define the Public Information

  MainWindow(unsigned long windowId); //Constructor for this class

virtual
   MainWindow();

```

```
static IColor
  penColorFromId              ( unsigned long Identifier ),
  fillColorFromId             ( unsigned long Identifier ),
  backColorFromId             ( unsigned long Identifier );

static unsigned long
  patternFromId               ( unsigned long Identifier ),
  penWidthFromId              ( unsigned long Identifier );

static IGraphicBundle::PenType
  penTypeFromId               ( unsigned long Identifier );

protected:

virtual Boolean
  setSize                     ( IMenuDrawItemEvent& evt,
                                ISize&              newSize ),
  draw                        ( IMenuDrawItemEvent& evt,
                                DrawFlag&           flag    ),

  command                     ( ICommandEvent&      event   );

private:                              //Define Private Information
  DrawingArea         drawingArea;    // move back to top of list.
  IToolBar            toolBar;
  IMenuBar            menuBar;
  IStaticText         infoText;
  IFlyText            flyText;
  IFlyOverHelpHandler flyOver;
  unsigned long       lastPenColorId;
  unsigned long       lastFillColorId;
  unsigned long       lastPenPatternId;
  unsigned long       lastFillPatternId;
  unsigned long       lastPenTypeId;
  unsigned long       lastPenWidthId;
  unsigned long       lastBackId;
  unsigned long       lastDrawOperationId;
  IToolBarButton      normalButton,
                      lineButton,
                      drawButton,
                      rectangleButton,
                      ellipseButton,
                      polylineButton,
                      polygonButton,
                      arcButton,
                      pieButton,
                      chordButton,
                      textButton,
                      bitmapButton;
};
⋮
```

In the .CPP file, we then do the following:

1. Initialize the main window on lines 56 through 90 and call the constructors for all the data members.
2. Set fly over help for the main window on lines 122 through 131.
3. Construct our drawing area on lines 173 through 186.
4. Define our current attribute bundle on lines 188 through 191.

5. Create the cursor for IGList to clean up the drawing area on lines 225 through 234. The cursor calls the destructor for each object in the IGList and deletes all the objects when the window is closed.

6. Handle mouse click events on lines 242 through 275.

7. Handle button messages. These events indicate a new graphic object is created of additional data points to add to an existing graphic object.

8. Set the draw state which determines how to interpret the mouse clicks.

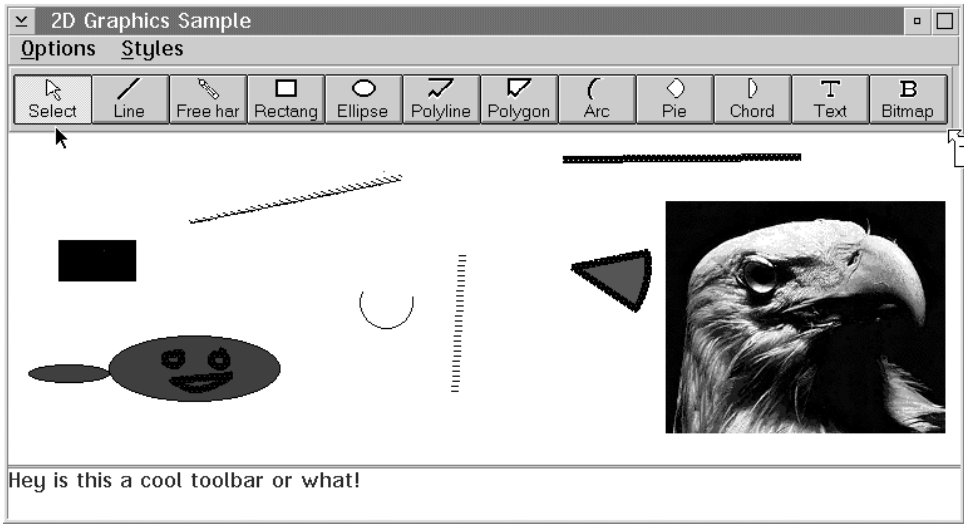
9. Create a line, rectangle, and ellipse on lines 356 through 378.

```
⋮
 51  /*----------------------------------------------------------------------------
 52  | MainWindow::MainWindow                                                      |
 53  |                                                                             |
 54  |                                                                             |
 55  ----------------------------------------------------------------------------*/
 56  MainWindow::MainWindow(unsigned long windowId)
 57    : IFrameWindow (                     //Call IFrameWindow constructor
 58      IFrameWindow::defaultStyle()       //  Use default plus
 59      | IFrameWindow::animated           //  Set to show with "animation"
 60      | IFrameWindow::menuBar            //  Frame has a menu bar
 61      | IFrameWindow::minimizedIcon,     //  Frame has an icon
 62      windowId),                         //  Main Window ID
 63      drawingArea( WND_DRAW, this, this ),
 64      toolBar( WND_TOOLBAR,this, IToolBar::aboveClient),
 65      menuBar( this, IMenuBar::wrapper ),
 66      infoArea( this, WND_TEXT ),
 67      flyText( 1054, &menuBar ),
 68
 69      // Set the initial delay for fly over help to 1 1/2 seconds and
 70      // set the regular delay to 1/3 seconds.
 71      flyOver( &flyText, &infoArea, 1500, 333 ),
 72      lastPenColorId( ID_COL_BLK ),
 73      lastFillColorId( ID_FLCOL_BLK ),
 74      lastPenPatternId( ID_PENPATTERN_SOLID ),
 75      lastFillPatternId( ID_FILLPATTERN_SOLID ),
 76      lastPenTypeId( ID_PENTYPE_SOLID ),
 77      lastPenWidthId( ID_PENWIDTH_1 ),
 78      lastDrawOperationId( ID_FILLANDFRAME ),
 79      normalButton(PALLET_NORM, &toolBar, &toolBar),
 80      lineButton(PALLET_LINE, &toolBar, &toolBar),
 81      drawButton(PALLET_DRAW, &toolBar, &toolBar),
 82      rectangleButton(PALLET_RECTANGLE, &toolBar, &toolBar),
 83      ellipseButton(PALLET_ELLIPSE, &toolBar, &toolBar),
 84      polylineButton(PALLET_POLYLINE, &toolBar, &toolBar),
 85      polygonButton(PALLET_POLYGON, &toolBar, &toolBar),
 86      arcButton(PALLET_ARC, &toolBar, &toolBar),
 87      pieButton(PALLET_PIE, &toolBar, &toolBar),
 88      chordButton(PALLET_CHORD, &toolBar, &toolBar),
 89      textButton(PALLET_TEXT, &toolBar, &toolBar),
 90      bitmapButton(PALLET_BITMAP, &toolBar, &toolBar)
 91  {
⋮
122    // Set the fly over help for the client window to the
123    // help information for the pointer (normal) button.
124
125    flyOver.setHelpText( drawingArea.handle(),
126                         IResourceId(0),
127                         IResourceId(PALLET_NORM + LONG_OFFSET ));
128
129    flyOver.handleEventsFor( this );
```

```
  flyOver.handleEventsFor( &drawingArea );
  flyOver.setLongStringTableOffset( LONG_OFFSET );

  setClient( &drawingArea );
  setFocus();
  show();
}

⋮
/*------------------------------------------------------------------------------
| DrawingArea::DrawingArea                                                     |
|                                                                              |
|                                                                              |
------------------------------------------------------------------------------*/
DrawingArea::DrawingArea( unsigned long windowId, IWindow* parent,
                          IWindow* owner, const IRectangle& initial )
  : IDrawingCanvas( windowId, parent, owner, initial ),
    gc(handle()),
    currentfont(),
    currentBundle(),
    currentBitmap(),
    dState( notDrawing ),
    currentObj( pointer ),
    iGraphic(0),
    moveGraphic(0),
    moveRect(IRectangle()),
    startingPt(), previousPt(), tempPt(),
    pointCount(0)
{
  currentBundle.setPenColor(IGraphicContext::defaultPenColor())
               .setFillColor(IGraphicContext::defaultFillColor())
               .setMixMode(IGraphicContext::defaultMixMode())
               .setDrawOperation( IGraphicBundle::fillAndFrame );

  gc.setMixMode( IGraphicBundle::xor ).setPenColor( IColor::white )
    .setFillColor( IColor::white )
    .setDrawOperation( IGraphicBundle::frame );
  setGraphicContext( &gc );

  setGraphicList( new IGList() );

⋮
  ((IMouseHandler*)this)->handleEventsFor(this);
}

/*------------------------------------------------------------------------------
| DrawingArea::DrawingArea                                                     |
|                                                                              |
|                                                                              |
------------------------------------------------------------------------------*/
DrawingArea:: DrawingArea( )
{
  // Delete all the graphic objects in the drawing canvas.
  IGList::Cursor graphicsCursor( *graphicList() );
  for ( graphicsCursor.setToFirst();
        graphicsCursor.isValid();
        graphicsCursor.setToNext() )
  {
    IGraphic* graphic(&(graphicList()->graphicAt(graphicsCursor)));
    delete graphic;
  } /* endfor */
  delete graphicList();
}

/*------------------------------------------------------------------------------
| DrawingArea::mouseClicked                                                    |
|                                                                              |
| Translate the mouse clicked events.                                          |
------------------------------------------------------------------------------*/
Boolean DrawingArea::mouseClicked( IMouseClickEvent& event )
{
  Boolean bRc = false;
  if ( event.mouseButton() == IMouseClickEvent::button1 &&
       event.mouseAction() == IMouseClickEvent::down )
  {
    button1Down(event.mousePosition());
    bRc = false;
  }
  else if ( event.mouseButton() == IMouseClickEvent::button1 &&
            event.mouseAction() == IMouseClickEvent::up )
  {
    button1Up(event.mousePosition());
    bRc = true;
  }
  else if ( event.mouseButton() == IMouseClickEvent::button1 &&
            event.mouseAction() == IMouseClickEvent::doubleClick )
  {
    button1DoubleClick(event.mousePosition());
    bRc = true;
  }
  else if ( event.mouseButton() == IMouseClickEvent::button2 &&
            event.mouseAction() == IMouseClickEvent::down )
  {
    button2Down(event.mousePosition());
  }
  else if ( event.mouseButton() == IMouseClickEvent::button2 &&
            event.mouseAction() == IMouseClickEvent::up )
  {
    button2Up(event.mousePosition());
  }

  return bRc;
}
⋮

/*------------------------------------------------------------------------------
| DrawingArea::button1Down                                                     |
|                                                                              |
| Handle button 1 down messages.  This event indicates a new graphic object is |
| to be created of additional data points to add to an existing graphic object.|
------------------------------------------------------------------------------*/
DrawingArea& DrawingArea::button1Down( const IPoint& point )
{
  switch (currentObj)
  {
    case pointer:
    {
      // Change all objects to the current pen and fill color.
      IGList::Cursor cursor( *graphicList(), gc, point );
      for (cursor.setToFirst(); cursor.isValid(); cursor.setToNext())
      {
        IGraphic& graphic(graphicList()->graphicAt( cursor ));
        this->refresh( graphic.boundingRect( gc ));//    .expandBy(1) );
        graphic.setGraphicBundle( currentBundle );
        graphic.drawOn( gc );
      } /* endfor */
    }
    break;
    case line:
      startingPt = point;
      previousPt = point;
      iGraphic = new IGLine( startingPt, previousPt );
      setDrawState();
    break;
    case rectangle:
      startingPt = point;
      previousPt = point;
      iGraphic = new IGRectangle(IRectangle(startingPt, previousPt));
      setDrawState();
    break;
    case ellipse:
      startingPt = point;
      previousPt = point;
      iGraphic = new IGEllipse( startingPt, 0L );
      setDrawState();
    break;
⋮
  } /* endswitch */
  return *this;
}

/*-----------------------------------------------------------------------------
| DrawingArea::button2Down                                                    |
|                                                                             |
| Determine the object under the mouse and start moving.                     |
-----------------------------------------------------------------------------*/
DrawingArea& DrawingArea::button2Down( const IPoint& point )
{
  if (drawState() == notDrawing)
  {
    moveGraphic = graphicList()->topGraphicUnderPoint( point, gc );
    if ( moveGraphic )
    {
      moveRect.setEnclosingRect(moveGraphic->boundingRect( gc ));
      previousPt = point;
      startingPt = point;
      capturePointer();
    }
  }
  return *this;
}

/*-----------------------------------------------------------------------------
| DrawingArea::mouseMoved                                                     |
|                                                                             |
| Handle button 1 down mouse move events.  This allows data points to be     |
| moved while the object is drawn with a rubber band effect.                 |
-----------------------------------------------------------------------------*/
Boolean DrawingArea::mouseMoved( IMouseEvent& event )
{
  IPoint point(event.mousePosition());
  if ( hasPointerCaptured() )
  {
    IRectangle windowRect(this->rect());
    windowRect.moveTo(IPoint(0,0));
    if (!windowRect.contains(point))
    {
      if ((short)point.x() < (short)windowRect.left())
        point.setX(windowRect.left());
      else if ((short)point.x() > (short)windowRect.right())
        point.setX(windowRect.right());
      else if ((short)point.y() < (short)windowRect.bottom())
        point.setY(windowRect.bottom());
      else if ((short)point.y() > (short)windowRect.top())
        point.setY(windowRect.top());

      IPoint mapPt( IWindow::mapPoint( point,
                                       this->handle(),
                                       IWindow::desktopWindow()->handle()));

      IWindow::movePointerTo( mapPt );
    }
  }

  // If we're not moving an object
  if (!moveGraphic)
  {
    if ( drawState() == drawing )
    {
      switch (currentObj)
      {
        case pointer:
        break;
        case line:
          ((IGLine*)iGraphic)->drawOn( gc );
          ((IGLine*)iGraphic)->setEndingPoint( point );
          ((IGLine*)iGraphic)->drawOn( gc );
        break;
        case rectangle:
        {
          IRectangle rc(((IGRectangle*)iGraphic)->enclosingRect());
          iGraphic->drawOn( gc );
          rc.sizeTo( rc.size() + point - previousPt );
          ((IGRectangle*)iGraphic)->setEnclosingRect( rc );
          iGraphic->drawOn( gc );
          previousPt = point;
        }
        break;
        case ellipse:
        {
          iGraphic->drawOn( gc );
          IPoint centerPt(((IGEllipse*)iGraphic)->enclosingRect().center());

          ((IGEllipse*)iGraphic)->setEnclosingRect(
            ((IGEllipse*)iGraphic)->enclosingRect().sizeTo( IPair(
                                                  abs(2*(point.x() - centerPt.x())),
                                                  abs(2*(point.y() - centerPt.y()))))
                                               .centerAt( centerPt ));
          iGraphic->drawOn( gc );
        }
        break;
:
      } /* endswitch */
    }
  }
  else
  {
    moveRect.drawOn( gc );
    moveRect.translateBy( point - previousPt );
    moveRect.drawOn( gc );
    previousPt = point;
  }
  return false;
}

/*------------------------------------------------------------------------------
| DrawingArea::button1Up                                                       |
|                                                                              |
| Handle button 1 up events.  This indicates a data points final location.     |
------------------------------------------------------------------------------*/
DrawingArea& DrawingArea::button1Up( const IPoint& point )
{
  if ( drawState() == drawing )
  {
    switch (currentObj)
    {
      case pointer:
```

```
      break;
      case line:
        ((IGLine*)iGraphic)->setEndingPoint( point );
        iGraphic->setGraphicBundle( currentBundle );
        iGraphic->drawOn( gc );
        setDrawState( notDrawing );
        graphicList()->addAsLast( *iGraphic );
      break;
      case rectangle:
      {
        IRectangle rc(((IGRectangle*)iGraphic)->enclosingRect());
        rc.sizeTo( rc.size() + point - previousPt );
        ((IGRectangle*)iGraphic)->setEnclosingRect( rc );
        iGraphic->setGraphicBundle( currentBundle );
        iGraphic->drawOn( gc );
        setDrawState( notDrawing );
        graphicList()->addAsLast( *iGraphic );
      }
      break;
      case ellipse:
      {
        IPoint centerPt(((IGEllipse*)iGraphic)->enclosingRect().center());

        ((IGEllipse*)iGraphic)->setEnclosingRect(
          ((IGEllipse*)iGraphic)->enclosingRect().sizeTo( IPair(
                                                abs(2*(point.x() - centerPt.x())),
                                                abs(2*(point.y() - centerPt.y()))))
                                             .centerAt( centerPt ));

        iGraphic->setGraphicBundle( currentBundle );
        iGraphic->drawOn( gc );
        setDrawState( notDrawing );
        graphicList()->addAsLast( *iGraphic );
      }
      break;
⋮
    } /* endswitch */
  } /* endif */
  return *this;
}

/*------------------------------------------------------------------------------
| DrawingArea::button2Up                                                       |
|                                                                              |
------------------------------------------------------------------------------*/
DrawingArea& DrawingArea::button2Up( const IPoint& point )
{
  if (moveGraphic)
  {
    moveRect.translateBy( point - previousPt );
    moveRect.drawOn( gc );
    moveRect.resetTransformMatrix();
    this->refresh( moveRect.boundingRect(gc).expandBy(1) );
    moveGraphic->translateBy( point - startingPt );
    moveGraphic->drawOn( gc );
    moveGraphic = 0;
    capturePointer(false);
  }
  return *this;
}

/*------------------------------------------------------------------------------
| DrawingArea::button1DoubleClick                                              |
|                                                                              |
| Handle button 1 up double click events.  In the case of polyline and polygon |
| a double click indicates the user has finished adding data points to the     |
| object.                                                                      |
-----------------------------------------------------------------------------*/
DrawingArea& DrawingArea::button1DoubleClick( const IPoint& point )
{
  if (drawState() == waitingForInput )
  {
    switch (currentObj)
    {
⋮
    }
  }
  return *this;
}

/*-----------------------------------------------------------------------------
| DrawingArea::setDrawState                                                    |
|                                                                              |
|                                                                              |
-----------------------------------------------------------------------------*/
DrawingArea& DrawingArea::setDrawState( const DrawState newState )
{
  dState = newState;
  if (dState == drawing)
  {
    if (!hasPointerCaptured())
      capturePointer();
  }
  else if (dState == notDrawing)
    capturePointer(false);
  return *this;
}
⋮
```

Figure 71 shows the compiled graphics sample.



Hey is this a cool toolbar or what!

Figure 71. Two-dimensional Graphic Sample

Graphics

43 Creating and Using Multimedia Controls

A multimedia application that integrates text and graphics with a combination of audio, motion video, images, and animation makes your application more attractive to the user, easier to use, and offers better mapping to real world objects.

You can use the window control classes and multimedia classes to implement an interface for your application that looks like the controls of common electronic devices, such as stereo components and video cassette recorders (VCRs). Your application can use these controls as interfaces to control audio and video media that is presented to the user. Keep in mind user expectations when you create your user interface. Mapping your user interface to the mental image the user has of real-world devices greatly enhances the ease of use of your product. For instance, most users are familiar with the play, stop, pause, fast forward, and reverse controls of an audio cassette recorder.

To use the multimedia classes, ensure that your working environment meets the following requirements:

Software IBM OS/2 V.3 Multimedia Presentation Manager/2 Version 1.1 is required for using the User Interface Class Library multimedia classes.

Hardware

If you are:	Hardware required:
Using MIDI, audio classes, or software motion video sound	OS/2-supported sound card
Using CDXA or AudioCD	CDROM
Hearing any sounds (CD or otherwise)	Speakers or headphones
Recording audio	Microphone

The following sections introduce multimedia concepts, describe multimedia devices and controls, and discuss the User Interface Class Library classes.

Understanding Multimedia

A *medium* is a carrier of information. A *multimedia computer system* is one that is capable of input or output on more than one medium. With the new class of computers, information in virtually any format can be combined into multimedia presentations.

Multiple types of input allow the user to interact with the computer in a style that best suits the information to be communicated, thus relieving overloaded input channels, such as a keyboard, mouse, or microphone.

Output information can be presented in more entertaining formats. Typically, output implies a computer display, video, or audio. Video has the potential to hold people's interest and illustrate concepts better than static images. Audio and speech contribute a unique quality to the multimedia system and can increase the information's content.

Understanding Multimedia Device Classes

The User Interface Class Library supports audio adapters, CD-ROM drives, video-disc players, logical devices, amplifier-mixers, and other hardware devices as media devices. These media devices are abstracted into classes that contain the data and functions essential for the operation of the real-world devices that they model.

The classes you define for your application combine the capabilities of several classes. Before defining the objects your application needs, choose real-world models that the user knows how to manipulate for the interfaces. You can then use the appropriate User Interface Class Library multimedia classes that provide the corresponding functions.

Understanding Base Device Classes

The base device classes let you create multimedia devices for your application. The following table lists the base classes and refers to the appropriate sections describing the multimedia devices. In addition, these sections describe how to use the devices.

You can directly instantiate device objects from the following classes:

Device Classes

Device	Class	For more information, refer to:
Audio amplifier-mixer	IMMAmpMixer	"Amplifier-Mixer Device (IMMAmpMixer)" on page 546
CD audio player	IMMAudioCD	"CD Audio Player Device (IMMAudioCD)" on page 550
CD Extended-Architecture player	IMMCDXA	"CD Extended-Architecture Player Device (IMMCDXA)" on page 579
Digital video player	IMMDigitalVideo	"Digital Video Player Device (IMMDigitalVideo)" on page 574
Master audio	IMMMasterAudio	"Master Audio Device (IMMasterAudio)" on page 545
MIDI sequencer	IMMSequencer	"MIDI Sequencer Device (IMMSequencer)" on page 556
Waveform audio player	IMMWaveAudio	"Waveform Audio Player Device (IMMWaveAudio)" on page 557

Understanding Abstract Device Classes

Common functions for devices are made available through *abstract device classes*. That is, abstract device classes allow inheriting classes to reuse common functions. Note that you cannot instantiate objects from these classes. The following sections describe the multimedia base class and abstract device classes. The multimedia base class (IMMDevice) is the parent class for the family of multimedia classes, including the base device classes and the other abstract classes.

Device	Purpose
IMMPlayableDevice	Used for many tasks, such as playing, pausing, and seeking.
IMMFileMedia	Used for devices that work with files.
IMMRecordable	Records, saves, cuts, pastes, allows undo, allows redo, saves-as.
IMMRemovableMedia	Opens and ejects media; unlocks and locks doors.

Using the User Interface Class Library Base Class for Multimedia

All of the multimedia device classes inherit from the IMMDevice class. This abstract class contains all of the common functions for device objects. These functions include the following:

- Querying the capabilities of a device
- Opening and closing devices
- Changing the speed and time formats

- Changing the audio (on or off)
- Controlling the volume

Playable Device Objects are usually instantiated from this class in applications that manage different types of devices (e.g. VCR and CD remote). You don't create an actual device rather the instantiated object is used to point to a device a user desires to activate (e.g. video player).

An object instantiated from IMMPlayableDevice is capable of performing tasks that a home device does to play such things as CDs or video tapes. In addition to the common device functions; like play, pause, seek for devices that support playback, IMMPlayableDevice objects are able to perform resume, stop, query position and length functions.

Creating and Using Audio Devices

This section introduces creating audio devices that play wave and musical instrument digital interface (MIDI) file formats.

Audio input and output is usually in the form of wave or MIDI files.

There is a distinction between sound and music; while it might not be distinctive to a radio, it is to a computer. Sound, such as the sound in wave files (see "Waveform Audio Player Device (IMMWaveAudio)" on page 557) is basically just digitized data that a computer cannot process. MIDI augments waveform audio by producing sound in the multimedia environment. Your system plays whatever is in a wave file out to your speakers. By comparison, music is actual information. The following section introduces basic concepts about these two formats.

Understanding MIDI Concepts

MIDI is a standardized set of data blocks or "messages" that instruct any MIDI-compatible sound source as to which notes to play. Rather than representing actual sound recordings, as a file of digitized audio does, a MIDI file merely describes what notes to play and includes settings for the sound or instrument, duration, stereo pan position (how far left or right), and volume.

A MIDI file is comprised of variable-length chunks. There are two types of chunks: a header chunk and one or more track chunks. The number of chunks are defined in the header. A MIDI event can be one of a number of things. It can be a message that turns a particular note on or off, that changes the voice being played by a particular channel, or that defines something about the piece being played.

When you create multimedia applications that play instrumental music, handle it with MIDI (music) rather than digitized audio files (sound). The relative size of the files involved is one of the best arguments for doing so. Developers never used to work with wave files because they could not handle the size. This might change now that compact discs (CDs) are common and hold large amounts of information.

Digital synthesis methods, either FM or wavetable playback, are customarily driven by MIDI.

MIDI files have the extension .mid and deliver more music per byte than other formats. MIDI files are comparable in size to ASCII text files, while the other music and sound formats (for example, wave) are comparable to color bitmaps. A digital audio recording of a musical instrument performance can consume megabytes of storage; a MIDI file describing that same performance can take only a kilobyte (K) or two. Wave and CD audio files can sometimes be too big to distribute easily, where as MIDI files are smaller. Compare 5 minutes of sound in a MIDI file to a 20 MB wave file. The MIDI file is about 10K while the wave file is 20 MB. There is a noticeable difference with your application's performance.

MIDI files do not support voice or words. The main role for MIDI in multimedia is music composition and production. Once the music is recorded, it can be played on a high-end synthesizer and recorded in wave or CD-audio formats.

Understanding Waveform Concepts

Waveform refers to digital representation of an original audio sound wave. *Audio* refers to sound waves that have a perceived effect on the human ear.

Digital recordings offer more consistency than MIDI files. A CD recording of music sounds virtually the same on any CD player you use, but a MIDI musical file could sound like, for instance, a French horn on one synthesizer and a kazoo on another. The sound depends on the quality of the sound card. However, MIDI music typically sounds cleaner, more realistic, and more professional than the digital recording, especially if you do not have a sound studio to record your tracks.

Wave files have the extension .wav and contain analog sound that has been recorded digitally. The pieces of sound are usually sampled sound stored as data. An *analog-to-digital converter* creates sampled sound. A wave file can reproduce sound with anything from telephone to compact disc quality in monaural or stereo under computer control.

Creating Audio Devices

The following sections describe audio devices and their related controls.

Master Audio Device (IMMasterAudio)

The master volume control determines the maximum volume level of all audio output devices in the system. It sets a scale by which all subsequent volume commands are based. For instance, if the volume control sets the master volume at 50 percent, then all volume levels are cut in half.

Use the IMMasterAudio class to create a master audio object.

The master audio object has functions that do the following:

- Returns the current or saved setting of headphones, speakers, and master volume.
- Saves the current setting of headphones, speakers, and master volume.
- Sets the headphones and speakers on or off.
- Sets the master volume to a percent of the total volume available.

Note: We recommend that you do not use a master volume control in your application unless it absolutely requires you to do so. This control affects all volumes in your system. One scenario that is appropriate for using a master volume control is when you are using the amplifier-mixer device class to control a complete system. In this case, you want to control all volume in the system.

Amplifier-Mixer Device (IMMAmpMixer)

The visuals and functions of the amplifier-mixer device are similar to the amplifier-mixer device on your home stereo system. Components are plugged into the amplifier-mixer so that audio signals can be transferred to a pair of attached speakers, headphones, or perhaps another device. The amplifier-mixer is the center of all audio signals and provides input or output switching and sound-shaping services, such as volume, treble, or base control.

Use the IMMAmpMixer class to create an amplifier-mixer device. Its specific functions include the following:

- Controlling the balance, bass, treble, gain, and pitch of a signal
- Connecting other devices that need the above functions to the amplifier-mixer
- Turning off and on the sound that is routed through the amplifier-mixer to another device

Creating an Amplifier-Mixer Device: An Example

An example that creates an amplifier-mixer device follows:

1. Define the device in the .hpp file.

```
class AmpHandler    : public ISliderArmHandler {
typedef ISliderArmHandler  Inherited;
//*********************************************************************
//Class:   AmpHandler                                                 *
//                                                                    *
// Purpose: Provide a handler for processing the circular sliders.    *
//          It is a subclass of ISliderArmHandler.                    *
//                                                                    *
//*********************************************************************
public:

 AmpHandler ();

 virtual Boolean  moving   (IControlEvent& evt);
};


class Amp    : public IMultiCellCanvas {
//*********************************************************************
// Class:   Amp                                                       *
//                                                                    *
// Purpose: Provide an  amplifier-mixer for use by             *
//          all of the devices.                                       *
//          It is a subclass of IMultiCell.                           *
//                                                                    *
//*********************************************************************
public:

  Amp(IMMAmpMixer* pAmp,
      unsigned long windowid,
      IWindow* parent,
      IWindow* owner);

  ICircularSlider
       slVolume,
       slBalance,
       slBass,
       slTreble,
       slPitch,
       slGain;

  IMMAmpMixer*  pAmpMixer;

  AmpHandler    ampHandler;

  IStaticText   name;
};

class MainWindow : public IFrameWindow {
//*****************************************************************
// Class:   MainWindow                                            *
//                                                                *
// Purpose: Main application window.                              *
//          It is a subclass of IFrameWindow.                     *
//*****************************************************************
public:
  MainWindow( unsigned long windowId);

private:
  ISetCanvas      clientCanvas;

  IMMWaveAudio*   wavPlayer;

  IMMAmpMixer*    ampMixer;

  Amp*            amp;
};
```

2. Create the constructors for the MainWindow and the amplifier-mixer.

```
⋮
/*-------------------------------------------------------------------
| MainWindow::MainWindow
--------------------------------------------------------------------*/
MainWindow::MainWindow( unsigned long windowId)
          : IFrameWindow(windowId),
            clientCanvas(CLIENTCANVASID,this,this)
{
   wavPlayer = new IMMWaveAudio(true);     // Create the wave player
                                           // The operating system
                                           // automatically
                                           // creates and associates an
                                           // amplifier-mixer with both
                                           // a wave device and CD device.
   ampMixer = new IMMAmpMixer(wavPlayer->connectedDeviceId(
                   IMMDevice::waveStream));
                                           // Create the amplifier-mixer
   amp        = new Amp(ampMixer, AMP_ID, &clientCanvas, this);
                                           // Create the layout canvas
⋮
/*------------------------------------------------------------------
| Amp::Amp
-------------------------------------------------------------------*/
Amp::Amp(IMMAmpMixer& aAmp,
         unsigned long windowid,
         IWindow*          parent,
         IWindow*          owner)
   : IMultiCellCanvas(windowid,parent,owner),
     name            (AMPNAMEID, this,owner),
     slVolume        (SL_VOLUME_ID,this,this,IRectangle(),
                      ICircularSlider::defaultStyle()
                      | ICircularSlider::proportionalTicks ),
⋮
   slVolume.setRange    (IRange(0,100)); // Customize the volume controls
   slVolume.setIncrement (IPair(10,1));
   slVolume.setText      ("Volume");
   addToCell             (&slVolume,     1, 1, 1, 1);
⋮
ampHandler.handleEventsFor(this);    // Handle events for the amplifier
}
```

3. Handle events.

```
⋮
/*----------------------------------------------------------------
| AmpHandler::moving
------------------------------------------------------------------*/
IBase::Boolean AmpHandler::moving   (
                                           IControlEvent& evt)
{
   Boolean result=false;
   Amp* pAmp = (Amp*)(evt.window());
   ICircularSlider* pSld = (ICircularSlider*)(evt.controlWindow());
   short val = pSld->value();

   if (pAmp->pAmpMixer)
   {
      switch (evt.controlId())
      {
       case SL_VOLUME_ID:          // Manage the volume control
            pAmp->pAmpMixer->setVolume(val);
            result = true;
            break;
       case SL_BALANCE_ID:        // Manage the balance control
        pAmp->pAmpMixer->setBalance(val + 50);
                    //Added 50 since balance can only be 1 to 100 where
                    //0 is full to the left and 100 is full to the right
            result = true;
            break;
       case SL_BASS_ID:          // Manage the bass control
            pAmp->pAmpMixer->setBass(val);
            result = true;
            break;
       case SL_TREBLE_ID:       // Manage the treble control
            pAmp->pAmpMixer->setTreble(val);
            result = true;
            break;
       case SL_PITCH_ID:                   // Manage the pitch control
            pAmp->pAmpMixer->setPitch(val);
            result = true;
            break;
       case SL_GAIN_ID:                            // Manage the gain control
            pAmp->pAmpMixer->setGain(val);
            result = true;
            break;
      } /* endswitch */
   } /* endif */

   return result;
}
```

Figure 72 shows an amplifier-mixer player. All audio signals and sound shapings are controlled here.

Audio Devices

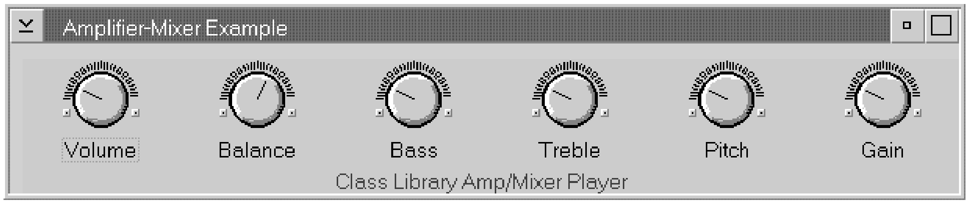


Figure 72. Amplifier-Mixer Example

CD Audio Player Device (IMMAudioCD)

The CD audio player device's interface should look and function similarly to your home CD system as it uses the same meduim, the compact disk.

A compact disc can store up to 74 minutes of 44.1-kilohertz, two-channel audio encoded as digital information. The audio compact disc, or the audio portion with both data and audio on it, is organized as tracks, where one track is typically one song. A track can be any length you want it to be, as long as it fits in the length of the disc. The length of a compact disc track is measured in minutes, seconds, and frames, where one frame is 1/75 second. It is possible to play portions of a track, starting and stopping within the accuracy of a single frame. While an application can play portions of a track, the amount of time required to seek from one track to another and locate the starting frame in question can be substantial; it can vary depending on where you are starting from. If your application calls for playing numerous sounds from a CD with precise timing, make sure they are located physically close together on the disc.

In addition to playing tracks, you can find out things about a CD, such as how many tracks it contains and how long each track is, or you can query a CD’s table of contents.

The CD device class, IMMAudioCD, provides access to devices that read CDs in order to play a compact disc’s digital audio data. This data format, which is digital audio, consists of sound that has been recorded as a sequence of 1s and 0s. A digital-to-analog converter recreates the original waveforms at playback.

You can perform the following functions with the IMMAudioCD class:

- Playing
- Scanning
- Tracking
- Querying a CD’s table of contents
- Querying a UPC code or country code (a UPC code is a serial number that is assigned to a disc)

- Setting up to play a particular song automatically
- Forwarding or reversing to a particular track
- Forwarding or reversing to a particular location (for example, 2 minutes into track 3)

Also useful is the ability to program the order in which a CD plays its tracks. Using the IProfile class, you can record in a file the song title or track information. The IMMAudioCD object loads the profile data and plays the CD based on the contents of the data. This might be useful in a CD store, where a system allows you to listen to CDs, traversing through the various tracks in six minutes.

You can use the following command to create a CD device object:

```
#include <immcdda.hpp>       // Define the header file

IMMAudioCD cdPlayer;         // Define the object

cdPlayer(true)               // Pass true to the device constructors so
                             // the devices are opened and no additional
                             // function calls are made before using
                             // the device.
```

Playing Audio Compact Discs

The interface for accessing and playing CDs should look very similiar to what you are used to on your home system. It should allow you to scan, play, stop, pause, and reverse. The IMMAudioCD class works only on discs that contain CD_DA tracks.

The application must ensure that the appropriate compact disc is in the CD drive. For example, a CD player application might simply update its track and time displays if a new disc is inserted and verified. If you try to open an IMMAudioCD object and there is not an audio CD in the CD-ROM drive, then the application sends a message that the medium is not valid. A CD-drive can only be accessed by one player at a time. An example of designing a user interface with a player panel containing a CD player device follows:

1. Define the device in the .hpp file.

```
⋮
class MainWindow;

class CD  : public IMultiCellCanvas,
             public ICommandHandler,
             public IObserver,
             public ISliderArmHandler,
             public ISelectHandler {
//**************************************************************
// Class:   CD                                                 *
//                                                             *
// Purpose: Provide a CD player.                               *
//                                                             *
//          It is a subclass of IMultiCell.                    *
//                                                              *
//                                                              *
//****************************************************************
public:

CD(   unsigned long     windowid,
      IWindow*          parent,
      IWindow*          owner);

protected:

virtual Boolean
  command( ICommandEvent& evt ),
  moving (IControlEvent& evt),
  selected(IControlEvent& event   );

virtual IObserver
  &dispatchNotificationEvent(const INotificationEvent&);

private:
IMMAudioCD
  cdPlayer;                                  // CD player device

IMMPlayerPanel
  baseButtons;                                // Player panel

IAnimatedButton                                // buttons to manipulate CD
  trackF,
  trackB,
  scanF,
  scanB,
  eject;

ICircularSlider
  volume;

IStaticText
 name,
 readout;

IRadioButton
  doorOpen,
  doorClosed;
};
⋮
class MainWindow : public IFrameWindow {
//***********************************************************
// Class:   MainWindow                                      *
//                                                          *
// Purpose: Main Window for the CD sample application.      *
//          It is a subclass of IFrameWindow.               *
//                                                          *
//***********************************************************
⋮
private:
CD*
  cd;
⋮
```

2. Create the main window and the CD

```
⋮
/*---------------------------------------------------------
| MainWindow::MainWindow
------------------------------------------------------------
MainWindow::MainWindow( unsigned long windowId)
          : IFrameWindow("Audio CD Example",windowId),
            clientCanvas(CLIENTCANVASID,this,this)
{
   cd       = new CD ( CD_ID, &clientCanvas, this);

⋮
/*----------------------------------------------------------
| CD::CD
------------------------------------------------------------
CD::CD( unsigned long windowid,
        IWindow*         parent,
        IWindow*         owner)
   : IMultiCellCanvas(windowid,parent,owner),
     readout     (READOUTID, this,this),
     name        (CDNAMEID, this, this),
     baseButtons (BASEBUTTONID, this,this, IMMDevice::audioCD),
     trackF      (TRACKFID,&baseButtons,&baseButtons,IRectangle(),
                  IWindow::visible | IAnimatedButton::animateWhenLatched),
     trackB      (TRACKBID,&baseButtons,&baseButtons,IRectangle(),
                  IWindow::visible | IAnimatedButton::animateWhenLatched),
     scanF       (SCANFID,&baseButtons,this,IRectangle(),
                  ICustomButton::autoLatch |
                  ICustomButton::latchable |
                  IWindow::visible | IAnimatedButton::animateWhenLatched),
     scanB       (SCANBID,&baseButtons,this,IRectangle(),
                  ICustomButton::autoLatch |
                  ICustomButton::latchable |
                  IWindow::visible | IAnimatedButton::animateWhenLatched),
     eject       (EJECTID,this,this,IRectangle(),
                  ICustomButton::autoLatch |
                  IWindow::visible | IAnimatedButton::animateWhenLatched),
     volume   (VOLID, this, this, IRectangle(),
                  ICircularSlider::defaultStyle() |
                  ICircularSlider::proportionalTicks),
     doorOpen (OPENBTN, this, this, IRectangle(),
                  IRadioButton::defaultStyle() |
                  IControl::group),
     doorClosed(CLOSEDBTN, this, this),
     cdPlayer()
{

⋮
   doorOpen.setText("Open");
   doorClosed.setText("Close door (if possible)");
   doorOpen.select();
   ISelectHandler::handleEventsFor(this);
   ICommandHandler::handleEventsFor(this);
   IObserver::handleNotificationsFor(cdPlayer);
   ISliderArmHandler::handleEventsFor(&volume);
}
```

3. Handle the CD track and scan buttons.

```
⋮
/*------------------------------------------------------------
| CD::command
-------------------------------------------------------------*/
IBase::Boolean CD::command( ICommandEvent& evt )
{
  Boolean rv  = false;
  switch ( evt.commandId() )
  {
  case TRACKBID:
       cdPlayer.trackBackward();
       rv=true;
       break;
  case TRACKFID:
       cdPlayer.trackForward();
       rc=true;
       break;
  case SCANFID:
       if (scanF.isLatched())
          cdPlayer.startScanningForward();
       else
          cdPlayer.stop();
       rv=true;
       break;
  case SCANBID:
       if (scanB.isLatched())
         cdPlayer.startScanningBackward();
       else
         cdPlayer.stop();
       rv=true;
       break;
  case EJECTID:
       if (cdPlayer.isMediaPresent())
          cdPlayer.openDoor();
       else
       {
          cdPlayer.closeDoor();
          if (cdPlayer.isMediaPresent())
            eject.unlatch();
       }
       rv = true;
       break;
    }
  return rv;
}
```

4. Handle the radio buttons to open and close the CD drive.

```
/*------------------------------------------------------------
| CD::selected                                    *
| Handle radio button click                                   *
-------------------------------------------------------------*/
IBase::Boolean CD::selected( IControlEvent& evt )
{
 Boolean rv = false;
 switch(evt.controlId())
  {
    case OPENBTN:
      // enable open cd
          cdPlayer.openDoor();
          rv = true;
      break;
    case CLOSEDBTN:
      // close cd
      cdPlayer.closeDoor();
      rv = true;
      break;
   }
  return rv;
}
```

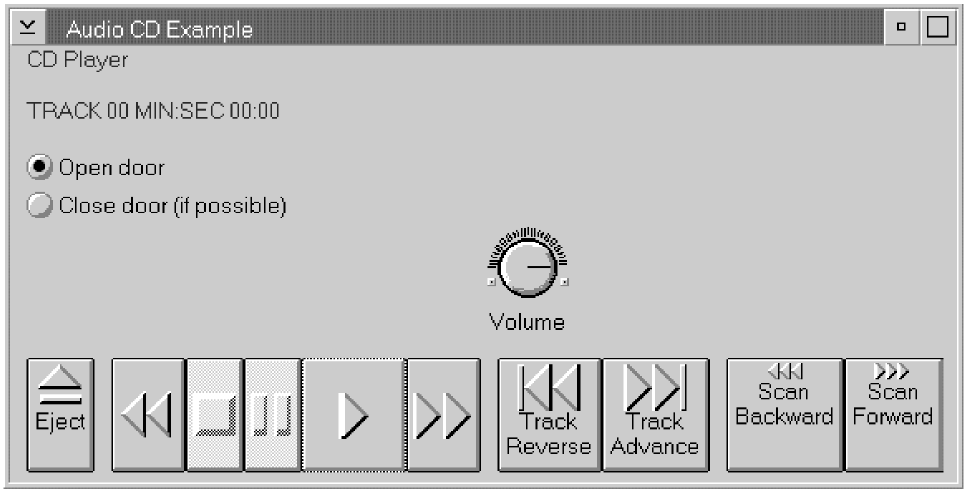
5. Handle the notification events.

```
/*-------------------------------------------------------------
| CD::displatchNotificationEvent
--------------------------------------------------------------*/
IObserver& CD::dispatchNotificationEvent(const INotificationEvent& event)
{
   if (event.notificationId() == IMMAudioCD::positionTimerId)
   {
      IMMTrackMinSecFrameTime* time =
         (IMMTrackMinSecFrameTime*)(event.eventData().asUnsignedLong());
      readout.setText(IString("TRACK ") +
                      IString(time->track()).rightJustify(2,'0') +
                      IString(" MIN:SEC ") +
                      IString(time->minutes()).rightJustify(2,'0') +
                      IString(":") +
                      IString(time->seconds()).rightJustify(2,'0'));
   } /* endif */
   else if (event.notificationId() == IMMAudioCD::trackStartedId)
   {
      IMMTrackMinSecFrameTime* time =
         (IMMTrackMinSecFrameTime*)(event.eventData().asUnsignedLong());
      readout.setText(IString("TRACK ") +
                      IString(time->track()).rightJustify(2,'0') +
                      IString(" MIN:SEC ") +
                      IString(time->minutes()).rightJustify(2,'0') +
                      IString(":") +
                      IString(time->seconds()).rightJustify(2,'0'));
   } /* endif */
   return *this;
}
```

6. Handle the volume slider events.

```
/*-------------------------------------------------------------
| CD::moving                                       |
--------------------------------------------------------------*/
⋮
IBase::Boolean CD::moving (IControlEvent& evt)
{
  Boolean result = false;
  ICircularSlider
   *pSld = (ICircularSlider*)( evt.controlWindow() );
  short
     val = pSld->value();
  switch(evt.controlId())
   {
     case VOLID:
      cdPlayer.setVolume(val);
      result = true;
      break;
    }
  return result;
}
```

Figure 73 shows an audio CD example. Note that the track information is displayed on the top part of the interface. The open and closed radio buttons control the CD door.



Audio CD Example

CD Player

TRACK 00 MIN:SEC 00:00

● Open door

○ Close door (if possible)

Figure 73. Audio CD Example

MIDI Sequencer Device (IMMSequencer)

A sequencer device plays a MIDI file by sending commands to a synthesizer where the commands are converted to the sounds of a specific musical instrument. The sequencer uses the timing commands to sequence the playing of the music.

Music devices with a sequencer, such as a Casio keyboard or a drum machine (a machine that reproduces percussion sounds), can record what is being played and can play what has been recorded previously. This recording is called a *sequence*. This sequence of music notes is stored in the MIDI format.

A *sequencer* is personal computer software that allows you to record, edit, and arrange multiple tracks of MIDI data. Most sequencers let you edit the messages in a sequence and link different sequences stored in memory. This finished sequence, ready for playback, is called a *song*. If you do not want to manipulate songs already recorded with a sequencer, you can also create original songs. A sequencer lets you record any style of music you want.

The MIDI sequencer device plays a MIDI song by sending commands from a MIDI file to a synthesizer where the commands are converted to the sounds of specific instruments. The IMMSequencer class is the base class for handling a MIDI sequencer device, and it supports the MIDI standard. Thus, the sequencer controls the characteristics of the MIDI information. In addition to allowing you to load MIDI files, the IMMSequencer class inherits all of the main functions, such as play, stop, pause, and record.

You can use the following command to create a MIDI sequencer object:

```
#include <immsequ.hpp>   // Define the header file

IMMSequencer midiPlayer; // Define the object

midiPlayer(true)         // Pass true to the device constructors so
                         // the devices are opened and no additional
                         // functions calls are made before using the
                         // device.
```

Waveform Audio Player Device (IMMWaveAudio)

The waveform audio device allows an application to play or record digital audio using files or application memory. Waveform audio devices require some form of input, that is, a file. The file contains the actual sound or waveform. The device can be opened with or without a file. If it is opened without a file, then a file is typically loaded later.

This device can use files or memory buffers. Buffering data improves performance of multimedia applications that perform numerous file input and output operations when accessing media devices. Applications that are performance-sensitive (that is, slow machines) can optimize file input and performance by buffering their data. If the data is already in the memory buffer, the operating system can transfer the record to the application's area without reading the sector from disk.

An object instantiated from IMMWaveAudio is capable of performing many tasks with a sound file. It can edit, play back, and record to name a few. In addition, the object inherits up the chain for the functions of play, stop, pause, and setFormat, plus cut, copy, and paste to and from a memory buffer.

You can use the following command to create a waveform audio device object:

Audio Devices

```
#include <immwave.hpp>   // Define the header file

IMMWaveAudio wavePlayer; // Define the object

wavePlayer(true)         // Pass true to the device constructors so
                         // the devices are opened and no additional
                         // functions calls are made before using
                         // the device.
```

Using Audio Devices

The following sections discuss various ways of using the audio devices previously introduced.

Playing a Waveform

If you want to design an application that can record and play back an audio signal from the user’s desktop, create an IMMWaveAudio object.

To play or record audio data, a device must be able to start and stop. Also, you might want fast-forward, reverse, and pause functions. Because most people are familiar with the control panel of the typical stereo, choose that model as your application’s user interface model.

An example that creates a user interface with an audio device and its controls follows:

1. Define the wave player device in the .hpp file.

```
⋮
class MainWindow;
class WavePlayer : public IMultiCellCanvas,
                     public ICommandHandler,
                     public ISliderArmHandler
{

public:
  WavePlayer(  unsigned long     windowid,
               IWindow*         parent,
               IWindow*         owner);
protected:
  virtual Boolean
      command( ICommandEvent& evt ),
      moving( IControlEvent& evt );

private:
  IStaticText               infoText;

  IAnimatedButton           playbtn,
                            stopbtn,
                            ffbtn,
                            rewbtn,
                            pausebtn,
                            recordbtn;

  ICircularSlider           volume;
  IMMWaveAudio           wavePlayer;
⋮
};
```

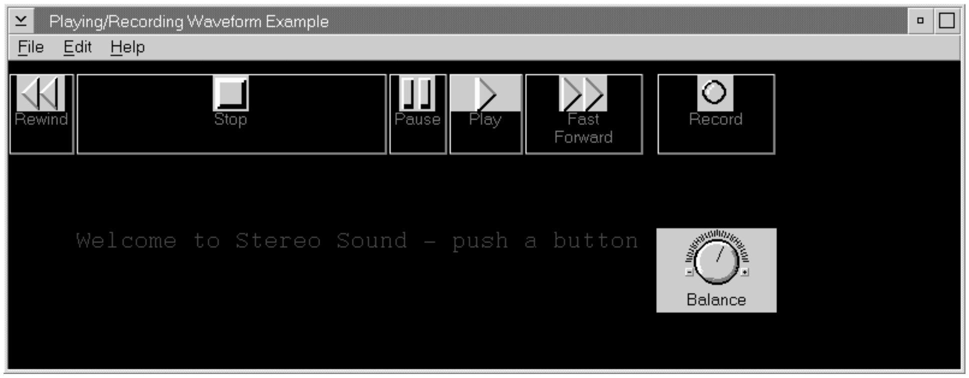
2. Create the main window and the wave player.

```
⋮
/*------------------------------------------------------------
| MainWindow::MainWindow
-------------------------------------------------------------*/
MainWindow::MainWindow( unsigned long windowId)
          : IFrameWindow("Playing/Recording Waveform Example",windowId),
            menuBar(windowId, this)

{
   wave = new WavePlayer(WAVEID, this, this);
   setClient( wave );
   sizeTo(ISize(550, 300));
   setFocus().show();
⋮
/*------------------------------------------------------------
| WavePlayer::WavePlayer
-------------------------------------------------------------*/
WavePlayer::WavePlayer( unsigned long windowid,
              IWindow*          parent,
              IWindow*          owner)
   : IMultiCellCanvas(windowid,parent,owner),
    volume(VOLID, this, this, IRectangle(),
         ICircularSlider::defaultStyle() |
         ICircularSlider::proportionalTicks),
⋮
     wavePlayer()
{

   volume.setArmRange     (IRange(0,100));
   volume.setRotationIncrement(10);
   volume.setText      ("Volume");
   volume.setValue( 100 );
   wavePlayer.setVolume( 100 );

⋮
   ICommandHandler::handleEventsFor(this);
   ISliderArmHandler::handleEventsFor( &volume );
}
```

3. Handle the player panel events.

```
/*------------------------------------------------------------
| WavePlayer::moving
-------------------------------------------------------------*/
IBase::Boolean WavePlayer::moving( IControlEvent& evt )
{
   Boolean
      result = false;

   ICircularSlider
     *pSld = (ICircularSlider*)( evt.controlWindow() );
   short
      val = pSld->value();
   switch( evt.controlId() )
   {
      case VOLID:
         wavePlayer.setVolume( val );
         result = true;
         break;
   }

   return result;
}

/*--------------------------------------------------------------
| WavePlayer::command
---------------------------------------------------------------*/
⋮
IBase::Boolean WavePlayer::command(ICommandEvent& evt)
{
  Boolean rv = false;
  switch (evt.commandId())
   {
     case MI_OPEN:
     {
        IFileDialog::Settings fdSettings;
        fdSettings.setTitle("Load file");
        fdSettings.setFileName("*.wav");
        IFileDialog fd(desktopWindow(), this, fdSettings);
        if (fd.pressedOK())
          wavePlayer.loadOnThread(fd.fileName());
        rv=true;
        break;
     }

     case PLAYID:
        wavePlayer.play();
        infoText.setText("Play Mode");
        rv=true;
        break;
     case STOPID:
        wavePlayer.stop();
        infoText.setText("Stop Mode");
        playbtn.unlatch();
        rv=true;
        break;
     case REWID:
        wavePlayer.seekToStart();
        infoText.setText("Rewind Mode");
        rv=true;
        break;
     case FFID:
        wavePlayer.seekToEnd();
        infoText.setText("FF Mode");
        rv=true;
        break;
     case PAUSEID:
        if (IMMDevice::paused == WavePlayer.mode())
          playbtn.latch();
        wavePlayer.pause();
        infoText.setText("Pause Mode");
        rv=true;
        break;
     case RECID:
      {
         wavePlayer.loadOnThread("hui.wav");
         recordbtn.latch();
         wavePlayer.record();
         infoText.setText( "Record Mode" );
         rv=true;
         break;
      }
    }
  return rv;
}
```

Figure 74 demonstrates a wave player interface.



Welcome to Stereo Sound - push a button

Figure 74. Wave Player Interface Example

Recording a Waveform

As stated earlier, a waveform is a digital representation of a sound wave. Different formats of a waveform, such as pulse code modulation (PCM), encode sound into digital data that can be sent to an amplifier-mixer device for subsequent conversion into audio. This signal can be played through conventional speakers or earphones.

The average waveform audio driver uses PCM, 22 kiloHertz, 16 bits-per-second, and monaural as the default for 16-bit adapters. If the adapter does not support 16-bit PCM, then the resolution (bits-per-second) is downgraded to 8 bits. The types of audio resolution are 8 (multimedia), 16 (CD audio) and 24 (high-end) digital bits-per-sample. *Red Book audio* is a music industry term technically known as the CD digital audio standard for music CD audio. *Yellow Book audio* is 16-bit or 8-bit digital audio played back by the sound card. Typically, yellow book audio is stored on the personal computer as .wav files.

Audio Devices

One of the typical uses of the waveform audio device is to digitize an input signal or sound into discrete samples for storage in a file. An example of this is recording an electronic audio mail message to tell someone about an idea, as opposed to typing a memo. An electronic audio mail application would provide the user with a simple control panel to allow the message to be recorded. Recording digitally means you get flawless sound quality that does not deteriorate.

You can record digital audio information in the format that fits your specific needs, such as for space or quality. For example, assume that a new wave audio file is created with the following command:

```
#include <immwave.hpp>
wavePlayer = new IMMWaveAudio(true);        // Create the object.
wavePlayer.record(10, 20);            // Enter begin and end time values.
```

When you create the file, you might want a file that is compatible with mu-law (the compression scheme used by a telephone system). The compression scheme can change the frequency range from a telephone to CD quality.

The attributes you need to consider when recording a file are the following:

- Format and its compression algorithm (pcm, adpcm, ibmcvcsd, okiadpcm, dviadpcm, digistd, digifix, or alaw)
- Bits per sample (16-bit is considered CD quality)
- Sampling rate (for example, 22 kiloHertz)
- Number of channels (stereo or monaural)

These attributes determine the audio quality. You can even make the decision to use low-bit resolution, a low sample rate, or even monaural versus stereo on the basis of disc space and bandwidth considerations. Always set the waveform format, sampling rate, resolution, and number of channels to ensure that the waveform is created with the desired parameters.

An example of code that sets these values follows:

```
#include <immwave.hpp>
wavePlayer = new IMMWaveAudio();          // Create the object.
wavePlayer.setBitsPerSample(Value);
wavePlayer.setSamplesPerSecond(Value);    // Set sampling rate
wavePlayer.setChannels(1);                // Monaural is 1 (stereo is 2)
```

Your application needs to define or select the recording source. The microphone is the default input device for recording waveforms.

The IMMWaveAudio class inherits the record function. An example of playing and recording a wave file follows:

1. Define the wave player device in the .hpp file.

```
⋮

class WavePlayer : public IMultiCellCanvas,
                     public ICommandHandler
                     public ISliderArmHandler,
                     public ISelectHandler
{
//****************************************************************
// Class:   WavePlayer                                           *
//                                                               *
// Purpose: Provide a WavePlayer for use by all of the devices.  *
//          It is a subclass of IMultiCell.                      *
//                                                               *
//****************************************************************
public:
  WavePlayer(  unsigned long     windowid,
               IWindow*          parent,
               IWindow*          owner);

   virtual Boolean
       command( ICommandEvent& event ),
       selected( IControlEvent& event ),
       moving( IControlEvent& event );

private:
  IStaticText                infoText;

  IAnimatedButton            playbtn,          // Player panel
                             stopbtn,
                             ffbtn,
                             rewbtn,
                             pausebtn,
                             recordbtn;

  ICircularSlider            volume;            // Volume control



  IRadioButton               mono,            // Radio Button controls
                             stereo;

  IStaticText                formatText;     //  Static Text

  IMMWaveAudio               wavePlayer;
};
⋮
//***************************************************************
// Class:   MainWindow                                          *
//                                                              *
// Purpose: Provide a WavePlayer for use by all of the devices. *
//          It is a subclass of IMultiCell.                     *
//                                                              *
//***************************************************************
class MainWindow : public IFrameWindow {

public:
  MainWindow( unsigned long windowId);


private:
      IMenuBar                 menuBar;
      WavePlayer               myWavePlayer;

};
```

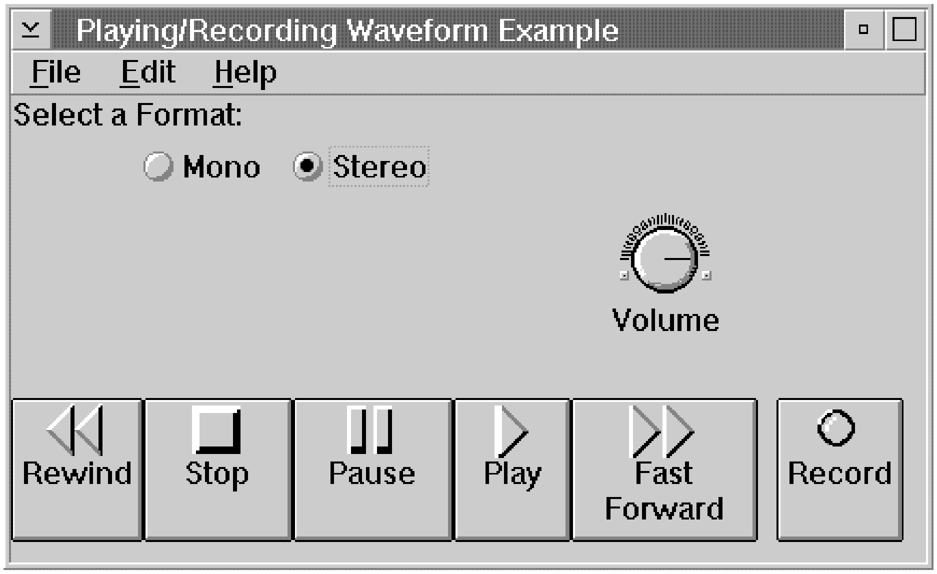
2. Create the main window and the wave player.

```
/*-----------------------------------------------------------
| MainWindow::MainWindow
-------------------------------------------------------------
MainWindow::MainWindow( unsigned long windowId)
          : IFrameWindow("Playing/Recording Waveform Example",windowId),
            menuBar(windowId, this),
            myWavePlayer( WINDOWID, this, this )

{

   setClient( &myWavePlayer );
   setFocus().show();
⋮
/*-----------------------------------------------------------
| WavePlayer::WavePlayer
-------------------------------------------------------------*/
WavePlayer::WavePlayer( unsigned long windowid,
              IWindow*          parent,
              IWindow*          owner)
     :IMultiCellCanvas  ( windowid, parent, owner ),
     volume            ( VOLID, this, this, IRectangle(),
                         ICircularSlider::defaultStyle() |
                         ICircularSlider::proportionalTicks ),
⋮
     infoText(INFOTXT, this, this),
     mono(MONOID, this, this, IRectangle(),
          IRadioButton::defaultStyle() | IControl::group),
     stereo(STEREOID, this, this),
     formatText(FORTEXTID, this, this),
     wavePlayer()
⋮
```

3. Handle events.

```
/*-----------------------------------------------------------
| WavePlayer::selected
-------------------------------------------------------------*/
IBase::Boolean WavePlayer::selected(IControlEvent& evt)
{
  Boolean rv = false;

  switch (evt.controlId())          // Handle the radio button controls
     {
     case MONOID:
      wavePlayer.setChannels(1);
      rv = true;
      break;
     case STEREOID:
      wavePlayer.setChannels(2);
      rv = true;
      break;
    }
   return rv;
 }
IBase::Boolean WavePlayer::moving( IControlEvent& evt )
{
   Boolean
      result = false;

   ICircularSlider
     *pSld = (ICircularSlider*)( evt.controlWindow() );
   short
      val = pSld->value();

   switch( evt.controlId() )
   {
      case VOLID:
         wavePlayer.setVolume( val );
         result = true;
         break;
   }
   return result;
}

IBase::Boolean WavePlayer::command( ICommandEvent& evt )
{
   Boolean
      rv = false;
   switch( evt.commandId() )
   {
      case MI_OPEN:
      {
         IFileDialog::Settings
            fdSettings;
         fdSettings.setTitle( "Load file" );
         fdSettings.setFileName( "*.wav" );
         IFileDialog
            fd( desktopWindow(), this, fdSettings );
         if ( fd.pressedOK() )
         {
            wavePlayer.loadOnThread( fd.fileName() );
         }
         rv=true;
         break;
      }
      case PLAYID:
      {
         wavePlayer.play();
         infoText.setText( "Play Mode" );
         rv=true;
         break;
      }
      case STOPID:
      {
         wavePlayer.stop();
         playbtn.enable();
         pausebtn.enable();
         ffbtn.enable();
         rewbtn.enable();
         infoText.setText( "Stop Mode" );
         playbtn.unlatch();
         rv=true;
         break;
      }
      case REWID:
      {
         wavePlayer.seekToStart();
         infoText.setText( "Rewind Mode" );
         rv=true;
         break;
      }
      case FFID:
      {
         wavePlayer.seekToEnd();
         infoText.setText( "FF Mode" );
         rv=true;
         break;
      }
      case PAUSEID:
      {
         if ( IMMDevice::paused == wavePlayer.mode() )
            playbtn.latch();
         wavePlayer.pause();
         infoText.setText( "Pause Mode" );
         rv=true;
         break;
      }
      case RECID:
      {
         recordbtn.latch();
         playbtn.enable( false );
         pausebtn.enable( false );
         ffbtn.enable( false );
         rewbtn.enable( false );
         wavePlayer.record();
         infoText.setText( "Record Mode" );
         rv=true;
         break;
      }
   }
   return rv;
}
```

Figure 75 demonstrates playing and recording a wave player. You can select a format and number of channels.



Select a Format:

○ Mono ● Stereo

Figure 75. Wave Player Playing and Recording Example

Loading the Audio or Video Device Data Files (IMMFileMedia)

There are three ways to load a file:

- Load with the wait calltype. The call does not return until the system loads the file into memory. This will tie up the operating system's windowing system until this call returns.
- Load with the nowait calltype. This creates a thread. The thread then loads the data and then notifies the attached observers when it is done. This call will return without waiting for the thread to complete. This will not tie up the operating system's windowing system.
- Use LoadOnThread. This creates a thread. The thread then loads the data. This call will not return until the thread finishes. This will not tie up the operating system's windowing system.

Using the Default Device Player (IMMPlayerPanel)

The interface for play, pausing, and stopping should appear similar to your system at home.

You can use VisualAge C++ Open Class Library's custom player panel. The IMMPlayerPanel class creates and manages a player panel. If you create the IMMPlayerPanel without passing in a device type then you will get the default buttons, which are:

- Play
- Stop
- Pause
- Rewind
- Fast forward

If you pass in an overlay, videoDisk animation, or digital video you will also get step forward and step backward buttons.

The base player panel is sufficient to control most multimedia devices.

The buttons are added to an IMultiCellCanvas in the following coordinates:

Play	4,1 or at 5,1 if the step buttons are enabled
Pause	3,1 or at 4,1 if the step buttons are enabled
fastForward	5,1 or at 6,1 if step buttons are enabled
rewind	1,1 or at 2,1 if step buttons are enabled
stop	2,1 or at 3,1 if step buttons are enabled
stepForward	7,1 if step buttons are enabled
stepBackward	1,1 if step buttons are enabled

An example of creating the VisualAge C++ Open Class Library custom player panel follows. See the other examples in this chapter for additional samples of using a player panel.

```
⋮

// Create a playable wave device
IMMWaveAudio player;

// Create the player panel and set it to control the wave audio player
IMMPlayerPanel panel(0x8008, &mainWindow, &mainWindow,
     IMMDevice::waveAudio);
panel.setPlayableDevice(player);
⋮
```

Figure 76 shows a custom player panel.

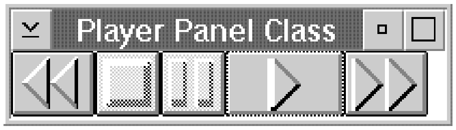


Figure 76. Player Panel Example

Note: The stop and pause buttons are disabled when starting up the application.

Editing a Waveform

The IMMWaveAudio class edits wave behavior. You can cut, copy, and paste to and from a memory buffer. A wave editor program allows you to record, edit, combine, and add special effects to a digital audio file. The file is not actually modified until the original file is saved. The editor allows you to mix tracks. You can use the musical editing process, for example, to correct mistakes in an artist's original interpretation or to change certain points of style before playback or final recording.

Using Save and Save As

The IMMRecordable class provides all the common behavior for devices that support recordable media. When you save a file the binary information is stored in addition to all of the wave's attributes. For example, if you are saving a waveform, some of the attributes that are saved follow:

- Sampling rate
- Resolution
- Waveform format
- Number of channels

Playing a Musical Instrument Digital Interface (MIDI) File

Typical user interfaces designed to offer playing of MIDI files have either a player panel containing a MIDI device object or a menu option to load a file via a file dialog or both. An example with a player panel follows:

1. Define the MIDI device in the .hpp file.

```
class MIDI  : public IMultiCellCanvas,
               public ICommandHandler,
               public ISliderArmHandler  {
//**********************************************************
// Class:   MIDI                                           *
//                                                         *
// Purpose: Provide a MIDI Player.                         *
//          It is a subclass of IMultiCell.                *
//                                                         *
//**********************************************************
public:

MIDI( unsigned long     windowid,
      IWindow*          parent,
      IWindow*          owner);

protected:
virtual Boolean
  command( ICommandEvent& evt ),
  moving (IControlEvent& evt);

private:

IMMSequencer
  midiPlayer;
IMMPlayerPanel
  baseButtons;

IAnimatedButton
  loadbtn,
  rec;

ICircularSlider
  volume;

IStaticText
 name;

};
⋮
```

2. Create the main window and the MIDI player.

```
⋮
/*-----------------------------------------------------------
| MainWindow::MainWindow
-------------------------------------------------------------*/
MainWindow::MainWindow( unsigned long windowId)
          : IFrameWindow   ( "Example MIDI Window", windowId ),
            clientCanvas   ( CLIENTCANVASID, this, this ),
            midi           ( MIDI_ID, &clientCanvas, this ),
            menuBar        ( windowId, this )
{
⋮
}
/*-----------------------------------------------------------
| MIDI::MIDI
-------------------------------------------------------------*/
MIDI::MIDI( unsigned long windowid,
        IWindow*          parent,
        IWindow*          owner)
   : IMultiCellCanvas(windowid,parent,owner),
     name        (MIDINAMEID, this, this),
     baseButtons (BASEBUTTONID, this,this),
     volume   (VOLID, this, this, IRectangle(),
                  ICircularSlider::defaultStyle() |
                  ICircularSlider::proportionalTicks),
     rec      ( RECID, &baseButtons, this, IRectangle(),
                ICustomButton::latchable |
                ICustomButton::latchable |
                IWindow::visible |
                IAnimatedButton::animateWhenLatched ),
     loadbtn   (LOADID, this, this, IRectangle(),
                     IWindow::visible |
                             IAnimatedButton::animateWhenLatched),
     midiPlayer()
{

   baseButtons.setPlayableDevice(&midiPlayer);   // Add button to panel
⋮
```

3. Handle events for the sliders.

```
/*------------------------------------------------------------
| MIDI::moving
--------------------------------------------------------------*/
IBase::Boolean MIDI::moving (IControlEvent& evt)
{
  Boolean result = false;

  ICircularSlider
    *pSld = (ICircularSlider*)( evt.controlWindow() );
  short
      val = pSld->value();
  switch(evt.controlId())
   {
     case VOLID:
      midiPlayer.setVolume(val);
      result = true;
      break;
    }
  return result;
}
```

4. Handle events for the radio buttons.

```
/*------------------------------------------------------------
| MIDI::command
--------------------------------------------------------------*/
IBase::Boolean MIDI::command(ICommandEvent& evt)
{
  Boolean rc = false;
  switch (evt.commandId())               // Load the midi file to play
  {
     case MI_OPEN:
     case LOADID:
        IFileDialog::Settings fdSettings;
        fdSettings.setTitle("Load file");
        fdSettings.setFileName("*.mid");
        IFileDialog fd(desktopWindow(), this, fdSettings);
        if (fd.pressedOK())
          midiPlayer.loadOnThread(fd.fileName());
        rc=true;

      case PLAYID:
      {
         midiPlayer.play();
         rv=true;
         break;
      }
    }
  return rv;
}
```

Figure 77 Displays a MIDI interface. Note that when you select the file menu option, a file dialog appears.

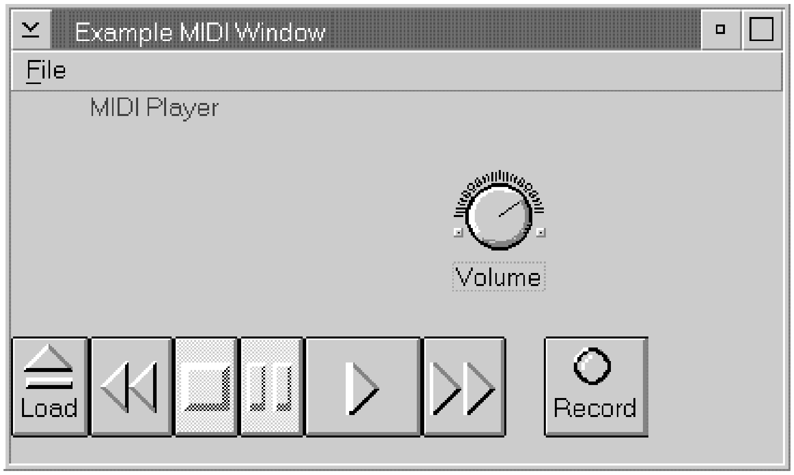


Figure 77. An example of a MIDI interface.

Using Animated Buttons

Animated buttons are customized push buttons. For example there is a play animated button which has a play button graphic on it. Use the IAnimatedButton class to create and manage the animated buttons. You can use a set of predefined bitmaps as graphics on the animated button (such as fast-forward or stop). The event handling for animated buttons is handled exactly as you handle pushbuttons. 🕮 See the wave player example in "Playing a Waveform" on page 558 for an example of animated buttons. The rewind, stop, pause, play, fast-forward and record buttons are all examples of animated buttons.

Using Circular Sliders

Use circular sliders for functions that a user can manipulate, such as volume and balance. You probably know how a linear slider works. A circular slider provides the same function; however, physically it looks different because it is circular. Like the dials found on your home electronics, its slider arm is shown as a radius of the dial. Outside the perimeter of the dial is a circular scale with tick marks representing incremental values the slider arm can point to. Its value can be tracked by pointing to any area on the dial and pressing the select button while moving the mouse on the desktop.

Because of its shape, the circular slider takes up less space than a linear slider and gives you more flexibility in designing a panel that has multiple controls.

An example of a creating a circular slider follows:

```
⋮
//*************************************************************
// main  - Application entry point
//*************************************************************
int main()
{
   IFrameWindow mainWindow ("Circular Slider Example",0x1000); // Create the frame window


   ICircularSlider slider(0x8008,             // Create the slider control
                          &mainWindow,
                          &mainWindow,
                          IRectangle(),
                          ICircularSlider::defaultStyle()
                          | ICircularSlider::proportionalTicks);

   slider.setArmRange(IRange(0, 100));         // Customize the circular slider
   slider.setRotationIncrement(1);
   slider.setText("Volume");


   mainWindow.setClient(&slider);
   ISliderArmHandler *shslider = new ISliderArmHandler();
   shslider->handleEventsFor(&slider);

   mainWindow.sizeTo(ISize(400,300));
   mainWindow.show();
   mainWindow.setFocus();
   IApplication::current().run();
                                              // Attach a handler to the control

   return 0;
}
⋮
```

Figure 78 shows a slider example. Note that its value is currently set at 30%.

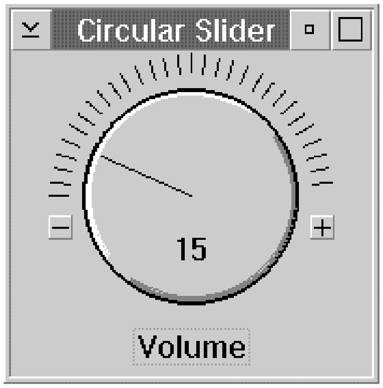


Figure 78. Circular Slider Example

Creating and Using Video Devices

The following section discusses creating a video device with controls that manipulate the device.

Understanding Video Concepts

Many people do not realize that the differences between analog and digital video are similar to the audio differences. An *analog* video is a series of squiggles that modulate a ray gun in a picture tube to paint images similar to what we see on a television.

Digitizing video freezes images into individual frames, each one a picture that can be manipulated. The frame rate is akin to the sample rate in that it explains how many times a second video is frozen. Frame rates generally vary from 12 (animation) to 24 (film) and up. Instead of bandwidth, a video's frame rate affects how smooth the motion of objects within the video image will appear.

Video resolution defines how much information is used to describe each dot or pixel of a frame. It ranges from 8 (multimedia) to 24 (higher-end graphics).

Creating Video Devices

The following sections introduce you to the video aspect of multimedia. You first learn how to create video devices.

Digital Video Player Device (IMMDigitalVideo)

A good way to visualize what a digital video player device does is to compare it to your VCR at home. Anything you can do with a VCR you can do with a digital video player device.

A digital video player device supports functions that manipulate digital video and audio files as well as digital video-only files. The audio-visual files have the extension of .avi (AVI means audio-visual interface).

The digital video device class, IMMDigitalVideo, includes functions that change the state of the window, query and set a device’s playback speed, and change a video window’s attributes. The video window is where the actual video is displayed. It can be free-floating or in a canvas window along with the buttons to manipulate the video. This class inherits functions, such as playback, record, query, adjust speed of motion video, and modify the audio attributes of the audio stored in the video file. A sound card is required to play back the sound part of the video files.

The system provided default video window does not provide a Close menu choice. If you want that capability, the digital video class provides the ability to replace the default video window with one of your choosing.

The command to create a digital video device object is:

```
#include <immdigvd.hpp>      // Define the header file

IMMDigitalVideo videoPlayer; // Define the object

videoPlayer(true)            // Pass true to the device constructors so
                             // the devices are opened and no additional
                             // functions calls are made before using
                             // the device.
```

Using Video Devices

The following sections discuss the intricacies of using video devices.

Playing Video Devices

The type of system in which you plan to play your video files is relative to the type of performance you will get when you play your files. The AVI files that store the clips can be rather large, even for something like a brief commercial. We recommend that you use the loadOnThread function when reading in large video clips. If you use load, it will more than likely tie up your OS/2 windowing system until the file is loaded. That can annoy your users. The loadOnThread function creates a thread to do the loading, which allows a user to continue doing an other task.

In addition to requiring a lot of storage, playing video files requires fast computers to run on. A 66-megahertz 486 system is an excellent choice. Lesser hardware produces less realistic movies with poorer resolution. Larger objects require more processor resources to animate. This means that it takes longer to move each frame of the animation from your hard drive, or wherever it resides, to your screen.

To sum up, factors affecting playback of a video are:

.

- Processing power of the CPU
- Throughput of data storage (for example, CD-ROM, hard disk, LAN)
- Efficiency of the display subsystem (such as the video adapter and display driver)

When a motion video device element is opened, the current position in the medium is the first playable area after any header or table of contents information.

Each frame in a motion video file has a number associated with it. The first frame is frame 0, the second frame is frame 1, and so on. The current position always indicates the frame that is about to be displayed. You can specify the play from and play to positions. You also give a frame position parameter to the seek command.

AVI files typically have digitized sound tracks along with their pictures. If you play the AVI file on a system with a sound card installed in it and turn on your speakers, you also hear the sound it contains.

Video Devices

The faster the machine, the faster the data processing and the playback are. Digital video is processor-intensive. Raw video requires huge amounts of memory—typically 900 kilobytes for a single frame of video, which equates to roughly 27 megabytes-per-second to record or play in real time. This is far beyond the capabilities of most personal computers today. So, digital video is often data-compressed to make it manageable. The simplest form of video compression is a smaller frame size or slower frame rate. This is the reason most digital video used in multimedia is so small and jerky. With the slow data transfer rate of CD drives, the video must be compressed further to be able to play it back. In sum: as your frame sizes and rates go up, so do your hardware requirements.

The tracks of data on a hard drive are laid out as concentric circles, whereas a CD has a single, spiral track, like an old phonograph record. Consequently, reading data requires more processing time on a CD.

One usually designs a player panel with push buttons to play video files. Video files can either be played in the application's window or in a free-floating window. A *free-floating window* displays a video in motion where the window is separate from the main application window. An example of doing either follows:

1. Define the video device.

```
⋮
class MainWindow : public IFrameWindow,
                           public ICommandHandler,
                           public ISelectHandler {
//**********************************************************
// Class:   MainWindow                                     *
//                                                         *
// Purpose: Main Window for MultiMedia sample              *
//     application.  It is a subclass of IFrameWindow.     *
//                                                         *
//**********************************************************
⋮
  ICanvas                  *videoCanvas;   // Define the canvas to place
                                           //   controls

  IMMDigitalVideo          videoPlayer;    // Define the video player

  IMMPlayerPanel           btnPanel;       // Define the player panel

  IAnimatedButton          loadBtn;        // Additional button
                                           // to load files

  IRadioButton             playFree,       // Radio buttons
                                           // to choose free-floating
                           playStatic;     // or static window
```

2. Create the main window.

```
/*-----------------------------------------------------------
| MainWindow::MainWindow
-------------------------------------------------------------*/
⋮
MainWindow::MainWindow( unsigned long windowId)
          : IFrameWindow(windowId),
            clientCanvas(CLIENTCANVASID,this,this),
                 btnPanel(PANELID, &clientCanvas, &clientCanvas),
            loadBtn(LOADID, &clientCanvas, &clientCanvas, IRectangle(),
                 IWindow::visible | IAnimatedButton::animateWhenLatched),
            playFree(FREEID, &clientCanvas, &clientCanvas, IRectangle(),
                 IRadioButton::defaultStyle() | IControl::group),
            playStatic(STATICID, &clientCanvas, &clientCanvas),
            videoPlayer(true)
{

   unsigned long size = 180;                  // Create and attach
                                              // the video canvas
                                              // to the main window
   videoCanvas = new ICanvas(VIDEOCANVAS, this, this);
   videoCanvas->setBackgroundColor(IColor::black);

   btnPanel.setPlayableDevice(&videoPlayer);   // Attach video
                                               // player to player panel
   loadBtn.setBitmaps(IAnimatedButton::eject);
   loadBtn.setText("Load");

   playFree.setText("Play video in floating window");
   playStatic.setText("Play video in static window");
   playFree.select();

   clientCanvas.addToCell(&btnPanel,   2, 7, 3, 1);
   clientCanvas.addToCell(&loadBtn,    1, 7);
   clientCanvas.addToCell(&playFree,   2, 2);
   clientCanvas.addToCell(&playStatic, 2, 4);
   addExtension(videoCanvas, IFrameWindow::aboveClient,
          size, IFrameWindow::thinLine);
```

3. Handle events.

```
⋮
/*-----------------------------------------------------------
| MainWindow::command
-------------------------------------------------------------*/
IBase::Boolean MainWindow::command(
                                                    ICommandEvent& evt)
{
 Boolean rv=false;

 switch (evt.commandId())            // Load the .avi file to play
  {
   case LOADID:
        IFileDialog::Settings fdSettings;
        fdSettings.setTitle("Load file");
        fdSettings.setFileName("*.avi");
        IFileDialog fd(desktopWindow(), this, fdSettings);
        if (fd.pressedOK())
          videoPlayer.loadOnThread(fd.fileName());
        rv=true;
```

```
          break;
     }
   return rv;
 }
 /*------------------------------------------------------------
 | MainWindow::selected
 --------------------------------------------------------------*/
 IBase::Boolean MainWindow::selected(
                                                  IControlEvent& evt )
 {
   Boolean rv = false;

   switch (evt.controlId())           // Handle radio buttons to switch
   {                                  // between playing free-floaing
     case STATICID:             // or statically on the frame window
        videoPlayer.setWindow(*videoCanvas);
        rv = true;
        break;
     case FREEID:
        videoPlayer.useDefaultWindow();
        rv = true;
        break;
      }
    return rv;
  }
```

Figure 79 shows a video device using a floating window.

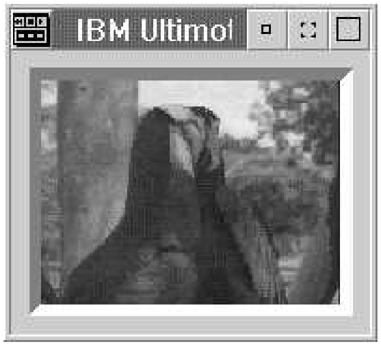
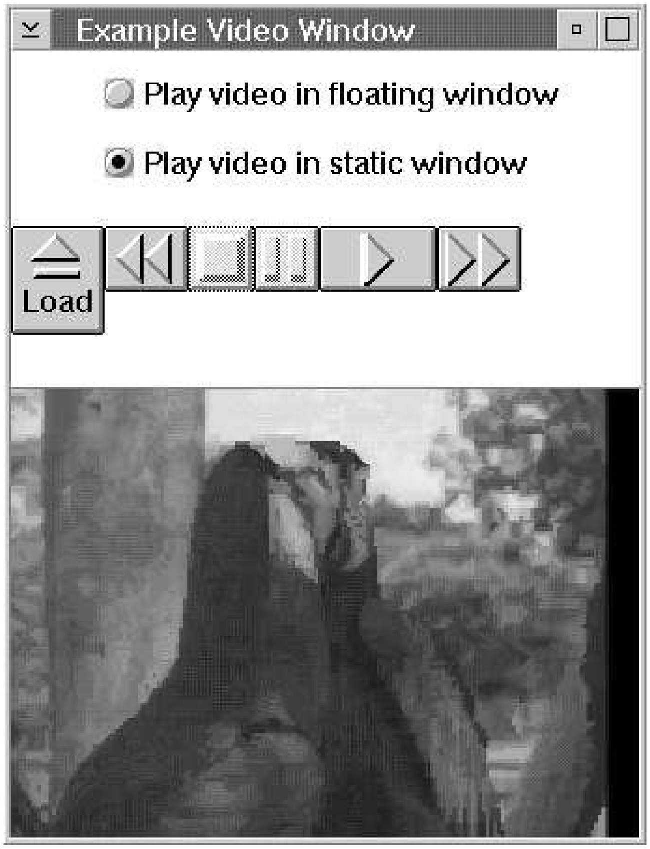


Figure 79. Motion Video Example

Figure 80 shows a video device displaying the video on a canvas child. Note that the video window is located in the same window as the video player panel.



Example Video Window
○ Play video in floating window
● Play video in static window
Load

Figure 80. Motion Video Example

CD Extended-Architecture Player Device (IMMCDXA)

The device class IMMCDXA provides access to devices that read CDs for the purpose of playing compact disc-extended architecture (CD-XA) data. CD-XA refers to a storage format that accommodates data that is stored in a mixture of formats. The CD-XA data is stored in part as files, in part as video, and in part as audio. The maximum amount of storage for each is:

Video	100 MB
Data	50 MB
Audio	20 MB

Playback control is managed by the CD-XA media device and the amplifier-mixer device.

An application for using the IMMCDXA class might be for video CDs or movie CDs. When giving a presentation, you might want to call up different data types at different times.

This class performs the same functions as the IMMAudioCD class.

Using Additional Multimedia User Interface Class Library Class Features

The following sections describe additional features provided by the User Interface Class Library:

- "Notifying Observer Objects" briefly discusses notifying objects when tasks are completed.
- "Controlling Position, Time, and Speed" describes the concepts of time, position, and speed in working with multimedia.
- The last section, "Multimedia Class Hierarchy" on page 582, presents the class hierarchy.

Notifying Observer Objects

The system returns an asynchronous response message (notification message) to the application to indicate events, such as completing a media device function or passing ownership of a media device from one process to another.

You can implement the notification architecture in your multimedia applications in the following ways:

- Call the INotifier::notifyObservers function with a user-defined event
- Subclass the IObserver class

See the Notification sections in the *Visual Builder User's Guide* and in the *Open Class Library Reference* for more information.

Refer to the CD sample for an example on using observer objects.

Controlling Position, Time, and Speed

Media position and time information are required as input and are also returned as output by many of the classes. All of these attributes—position, time, and speed—are relative in terms of time.

Time

There are 8 different time classes:

- IMMTime
- IMMillisecondTime
- IMMHourMinSecFrameTime

- IMM24FramesPerSecondTime
- IMM25FramesPerSecondTime
- IMM30FramesPerSecondTime
- IMMMinSecFrameTime
- IMMHourMinSecTime

There are various types of time formats (for example, milliseconds, HourMinSec, 24FramePerSecond). Time formats vary, depending on the device being used and the format of the data being operated on.

Descriptions of the time classes are as follows:

Class	Description
IMMTime	The base device time class. It provides behavior common to all device times. MMTime is the standard time and media position format supported by the media control interface. The time unit is 333 microseconds.
IMMMillisecondTime	Represents one-thousandth of a second.
IMMHourMinSecFrameTime	Represents the hours-minutes-seconds-frames time format.
IMM24FramesPerSecondTime IMM25FramesPerSecondTime IMM30FramesPerSecondTime	Represent the frame-numbering system developed by the Motion Picture and Television engineers that assigns a number to each frame of video. The 8-digit code is in the form HH:MM:SS:FF (hours, minutes, seconds, frame number). The numbers track elapsed hours, minutes, seconds, and frames from any chosen point. The classes differ only in their respective frames-per-second format. They are 24, 25 and 30 frames-per-second respectively. These packed formats represent elapsed hours, minutes, seconds, and frames from any specified point.
IMMMinSecFrameTime	Represents the time format based on the 75 frames-per-second CD digital audio standard.
IMMHourMinSecTime	Represents the hours-minutes-seconds time format.

The time classes contain methods that manipulate (add, subtract) and parse the formatted time data.

Position Position relates to time. For example, you might want to seek 3 minutes into a song on a compact disc. Position is the number of units before the request is executed. Units can be bytes, time, or some other unit of measure.

Speed Speed can be thought of in two ways: *framesPerSecond* and as a percentage. The percentage value refers to a percent of the maximum speed for the device. You might use a percentage when you currently are playing video at the device's fastest rate of one hundred percent. You could cut the rate to 60 percent. To do that, create a IMMSpeed object with a parameter of 60 and pass it to the set speed function. Or, you could use frames-per-second if you want to play back a video at specific frames-per-second.

The tracks of data on a hard drive are laid out as concentric circles, whereas a CD has a single, spiral track, like an old phonograph record. Consequently, reading data requires more processing time on a CD.

Multimedia Class Hierarchy

The names of the multimedia classes provide information about the data and functions available to you for your application.

In addition to their own data and functions, the multimedia classes can access the data and functions of the classes they inherit from. You can derive specialized classes for your application from this extensive hierarchy.

See "Multimedia Classes" on page 664 for the complete multimedia hierarchy.

Multimedia Samples

The purpose of the mmremote and mmstereo sample programs is to illustrate how to use the User Interface Class Library multimedia classes to create multimedia applications. Although the samples do not demonstrate all of the multimedia controls and functions, they do provide you with examples that span the most common uses of the classes. You can modify and use the sample programs as templates to start construction of your own multimedia applications. The multimedia samples are located in the following directory: \ibmcpp\samples\ioc

Subdirectory Structure

Code for the sample programs and the associated files you need to understand and run the samples are located in the following subdirectories of the sample directory:

Sample	Subdirectory
Stereo Control Panel	mmstereo
Universal Remote Control	mmremote

44 Providing Help Information

Help information is the information about how to use an application. By describing an application's choices, objects, and interaction techniques, help information can assist users in learning to use a product.

The User Interface Class Library provides an IHelpWindow class that uses the OS/2 Information Presentation Facility (IPF) to provide help information for applications. You should create and associate an IHelpWindow object with one of your application's main windows. The User Interface Class Library also provides an IHelpHandler class to process help window events. When you associate an application window with a help window, help events are dispatched to the help handlers attached to the application window.

Creating Help Information

Use the following steps to create help information for your application:

1. Create a file containing the help information.

   Create the source text that displays in your application's help window using the IPF format (.IPF file). Compile your IPF file into a help file (.HLP file) using the IPFC compiler. Refer to the *OS/2 Information Presentation Facility Guide and Reference* for descriptions of the tags you use to create the source .IPF file.

   For an example of an IPF source file, refer to the Hello World version 5 AHELLOW5.IPF file, which is described in Chapter 50, "Adding Split Canvases, a List Box, Native System Functions, and Help" on page 635.

2. Define the help window title and the help submenu in your resource file. In Hello World version 5, the help window title and help submenu are defined in the AHELLOW5.RC file, as follows:

   ```
   ⋮
   STR_HTITLE, "C++ Hello World - Help Window" //Help window title string
   ⋮
   SUBMENU "~Help", MI_HELP, MIS_HELP          //Help submenu
     BEGIN
       MENUITEM "~General help...",   SC_HELPEXTENDED, MIS_SYSCOMMAND
       MENUITEM "~Keys help...",      SC_HELPKEYS, MIS_SYSCOMMAND
       MENUITEM "Help ~index...",     SC_HELPINDEX, MIS_SYSCOMMAND
     END
   ⋮
   ```

   MI_HELP is the help menu ID.

Normally, you specify MIS_HELP for a menu item to cause a help event, rather than a command event, to be posted when the menu item is selected. OS/2 PM ignores MIS_HELP specified on submenu items.

When MIS_SYSCOMMAND is specified with the predefined SC_HELP* IDs, a system command event is generated. The default system command handler recognizes the predefined IDs and shows the appropriate help panel, except for SC_HELPKEYS, which by default does nothing. You can override this default processing for SC_HELPKEYS, using an IHelpHandler, which is described in a later step.

3. Define a help table in the resource file.

   The help table defines the relationship between the window ID and the general or contextual panel ID that is defined in the IPF file. The following help table is defined in the resource file, AHELLOW5.RC, for Hello World version 5:

```
HELPTABLE HELP_TABLE
  BEGIN
    HELPITEM WND_MAIN,       SUBTABLE_MAIN,   100
    HELPITEM WND_TEXTDIALOG,  SUBTABLE_DIALOG, 200
  END

HELPSUBTABLE SUBTABLE_MAIN                   //Main window help subtable
  BEGIN                                      //
    HELPSUBITEM WND_HELLO, 100               //Hello static text help ID
    HELPSUBITEM WND_LISTBOX,102              //List box help ID
    HELPSUBITEM MI_EDIT, 110                 //Edit menu item help ID
    HELPSUBITEM MI_ALIGNMENT, 111            //Alignment menu item help ID
    HELPSUBITEM MI_LEFT, 112                 //Left command help ID
    HELPSUBITEM MI_CENTER, 113               //Center command help ID
    HELPSUBITEM MI_RIGHT, 114                //Right command help ID
    HELPSUBITEM MI_TEXT, 199                 //Text command help ID
  END                                        //

HELPSUBTABLE SUBTABLE_DIALOG                 //Text dialog help subtable
  BEGIN                                      //
    HELPSUBITEM DID_ENTRY, 201               //Entry field help ID
    HELPSUBITEM DID_OK, 202                  //OK command help ID
    HELPSUBITEM DID_CANCEL, 203              //Cancel command help ID
  END                                        //
```

   WND_HELLO and WND_LISTBOX are control IDs, MI_* are menu item IDs, and the DID_* are push button IDs. Each window ID is related to a help panel ID. In the preceding example, WND_MAIN and WND_HELLO both correspond to help panel ID 100. That is, pressing the **F1** key in the main window area displays the same help panel as selecting **General help...** from the **Help** submenu.

4. Create a help window object for your application window.

   Use the IHelpWindow class to associate help information with an application window. Hello World version 5 defines the private data member, helpWindow, as an IHelpWindow object. It is initialized in the AHelloWindow constructor in AHELLOW5.CPP AHELLOW5.HPP.

```
⋮
class AHelpHandler : public IHelpHandler {
⋮
protected:
virtual Boolean
     keysHelpId(IEvent& evt);
};
⋮
```

5. Provide the overridden virtual function keysHelpId, which is called when keys help is requested. The following code, from the Hello World version 5 AHELLOW5.CPP file, shows how to implement this function.

   ```
   ⋮
   IBase::Boolean AHelpHandler :: keysHelpId(IEvent& evt)
   {
     evt.setResult(1000);                //1000=keys help ID in
                                         //  ahellow5.ipf file

     return (true);                      //Event is always processed
   } /* end AHelpHandler :: keysHelpId(...) */
   ⋮
   ```

   In the preceding code, the help panel ID for the Hello World version 5 keys help is set in the event result.

6. Start and stop help events processing.

   Your help handler, previously described, does not begin handling help events until the you use the handleEventsFor member function. For example, the following code causes the helpHandler to begin processing help events for this frame window:

   ```
   ⋮
   helpHandler.handleEventsFor(this);
   ⋮
   ```

   Typically, you include this statement in the constructor for the frame window.

   KeyConcept.Note that the window which handles help events must be an associated window. That is, you should identify the window as the associated window on the IHelpWindow constructor or explicitly identify the window as an associated window using the IHelpWindow::setAssociatedWindow function.

   When you want to stop handling help events, for example, when you close your frame window, use the stopHandlingEventsFor member function, as follows:

   ```
   ⋮
   helpHandler.stopHandlingEventsFor(this);
   ⋮
   ```

   You typically include this statement in the destructor for the frame window.

7. Associate secondary frame windows with the parent window's help window.

   You can use an owner window's help window for secondary frame windows by using the IHelpWindow::setAssociatedWindow member function. This function adds the secondary window to the help event chain for a specific help window. Specify a pointer to the secondary window as the one argument to this function.

   In many cases, you will want to make this association in the constructor of the secondary frame window, but you will not be passed a pointer to the owner window's help window. To get a reference to the owner's help window, use the static IHelpWindow member function helpWindow, specifying the owner window as the argument.

   Hello World version 5 provides an example in the ADIALOG5.CPP file, as follows:

   ```
   ⋮
   IHelpWindow::helpWindow(ownerWnd)->setAssociatedWindow(this);
   ⋮
   ```

8. Attach the following special handler to child frame windows in your application. This handler is needed so that help processes correctly for these windows.

   ```
   class ChildFrameHelpHandler : public IHandler {
   typedef IHandler Inherited;
   /*******************************************************************************
   *  This handler enables the OS/2 Help Manager to use help tables to display    *
   *  contextual help for a child frame window (one whose parent window is not    *
   *  the desktop).  This handler should only be attached to child frame windows. *
   *******************************************************************************/
   public:
   virtual ChildFrameHelpHandler
    &handleEventsFor       ( IFrameWindow* frame ),
    &stopHandlingEventsFor ( IFrameWindow* frame );
   protected:
   virtual Boolean
     dispatchHandlerEvent  ( IEvent& evt );
   ChildFrameHelpHandler
    &setActiveWindow       ( IEvent& evt, Boolean active = true );
   private:
   virtual IHandler
    &handleEventsFor       ( IWindow* window  ),
    &stopHandlingEventsFor ( IWindow* window  );
   };

   IBase::Boolean ChildFrameHelpHandler :: dispatchHandlerEvent ( IEvent& evt )
   {
     switch ( evt.eventId() )
     {
        case WM_ACTIVATE:
           setActiveWindow(evt, evt.parameter1().number1());
           break;
        case WM_HELP:
           setActiveWindow(evt, true);
           break;
        default:
   ```

```
        break;
  } /* endswitch */

  return false;                    // Never stop processing of event
}

ChildFrameHelpHandler&
  ChildFrameHelpHandler :: setActiveWindow ( IEvent& evt,
                                             Boolean active )
{
  IHelpWindow* help = IHelpWindow::helpWindow(evt.window());
  if (help)
  {
     IFrameWindow* frame = 0;
     if (active)
     {
        frame = (IFrameWindow*)evt.window();
     }
     help->setActiveWindow(frame, frame);
  }
  return *this;
}

ChildFrameHelpHandler&
  ChildFrameHelpHandler :: handleEventsFor ( IFrameWindow* frame )
{
  IASSERTPARM(frame != 0);
  Inherited::handleEventsFor(frame);
  return *this;
}

ChildFrameHelpHandler&
  ChildFrameHelpHandler :: stopHandlingEventsFor ( IFrameWindow* frame )
{
  IASSERTPARM(frame != 0);
  Inherited::stopHandlingEventsFor(frame);
  return *this;
}

IHandler& ChildFrameHelpHandler :: handleEventsFor ( IWindow* window  )
{          // private to hide version in IHandler
  ITHROWLIBRARYERROR(IC_MEMBER_ACCESS_ERROR,
                     IErrorInfo::invalidRequest,
                     IException::recoverable);
  return *this;
}

IHandler& ChildFrameHelpHandler :: stopHandlingEventsFor ( IWindow* window  )
{          // private to hide version in IHandler
  ITHROWLIBRARYERROR(IC_MEMBER_ACCESS_ERROR,
                     IErrorInfo::invalidRequest,
                     IException::recoverable);
  return *this;
}
```

Refer to the *Open Class Library Reference* for more information on IHelpWindow and IHelpHandler.

Adding Fly Over Help

Fly over help displays short help windows for the object that the mouse pointer is currently positioned over. As users move their mouse pointer over various objects, different help windows are displayed. In addition, you can display descriptive text for the object in a text control, such as the information area at the bottom of the window.

You can use the following classes to add fly over help to your applications:

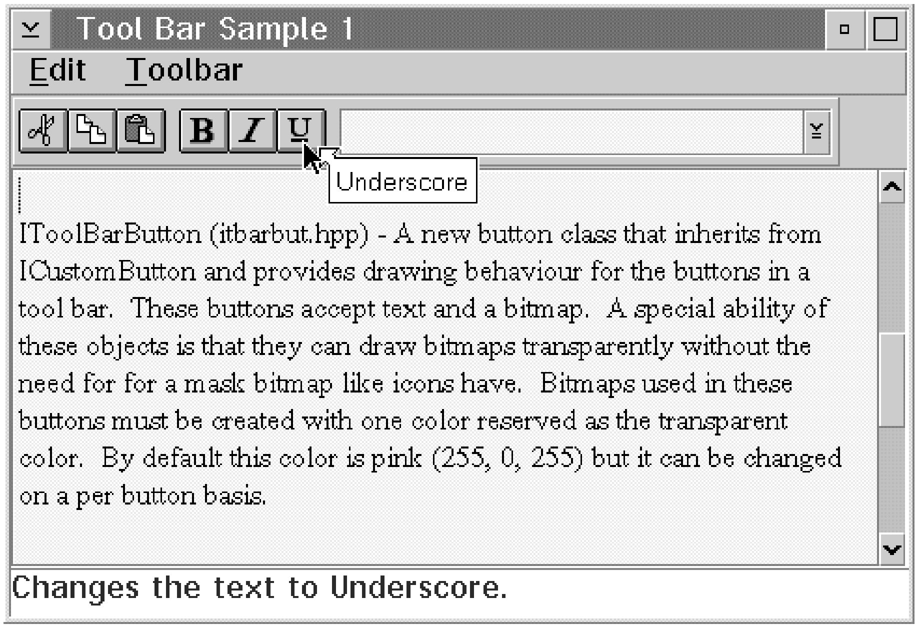
IFlyText Use this static text control to display brief informative messages for a window, such as the function of a push button contained in a tool bar.

ITextControl Use a class derived from this base class to display longer, more detailed text. Typically, you use an information area as the ITextControl.

IFlyOverHelpHandler Attach an IFlyOverHelpHandler to a window to provide context-specific help messages for any window that is a child of the window you attached the handler to.

For more information on these classes, see the *Open Class Library Reference*.

Figure 81 shows an example of fly over help used on a tool bar.



IToolBarButton (itbarbut.hpp) - A new button class that inherits from ICustomButton and provides drawing behaviour for the buttons in a tool bar. These buttons accept text and a bitmap. A special ability of these objects is that they can draw bitmaps transparently without the need for for a mask bitmap like icons have. Bitmaps used in these buttons must be created with one color reserved as the transparent color. By default this color is pink (255, 0, 255) but it can be changed on a per button basis.

Changes the text to Underscore.

Figure 81. Example of Fly Over Help on a Tool Bar

Displaying Fly Over Help Information

The IFlyText control displays help messages in a bordered window that is sized large enough to contain the help text. The text is displayed using one row that does not reflow. The default font used to display the help messages is an 8-point Helvetica.

In addition to displaying the help message in a window with a border around it, IFlyText controls draw an arrow pointing outward from one of the four corners of the control. This arrow points to the window for which the help message is being displayed.

The IFlyText control positions itself relative to one of the corners of the window the mouse pointer is over. Fly over help displays the help message so that none of the text is drawn outside the desktop. This determines the corner of the IFlyText control from which the arrow is drawn. The order in which the IFlyText control attempts to position itself is as follows:

1. Lower-right corner
2. Lower-left corner
3. Upper-left corner
4. Upper-right corner

Fly Over Help

You can construct instances of this class from a given window ID and an owner window.

For example:

```
IFrameWindow       frameWindow( 0x1000 );
IToolBar           toolbar( 0x1001, &frameWindow, &frameWindow );
IFlyText           flyText( 0x1002, &toolbar );
IFlyOverHelpHandler flyOverHelpHandler( &flyText );

flyOverHelpHandler.handleEventsFor( &toolbar );
```

IFlyText provides the function setText to set the accessible attributes of each instance, and provides functions setRelativeWindowRect and relativeWindowRect to set and query the position relative to the IFlyText control.

Attaching Handlers to Provide Context-Sensitive Help

By attaching an IFlyOverHelpHandler to a window, you can provide context-specific help messages for any window that is a child of the window you attached the handler to.

You can use the IFlyOverHelpHandler to update an IFlyText control, an ITextControl, or both. The IFlyText control contains short messages (one or two-words, for example) for a window and ITextControl displays more descriptive text in the information area. You do not need a string associated with every window in your application. When a help string cannot be found for a window, a single blank is displayed by default to keep the frame extension handler from hiding the ITextControl when it contains a null string. You can also use the setMissingText function to set the text to be displayed when an information string cannot be found.

Note: New-line characters are removed from a string before they are displayed in the IFlyText control.

The last two parametera of each handler constructor is a time delay expressed in milliseconds. The first delay, indicates the time the mouse pointer must remain in in the same location before the fy over help is displayed for the first time. The second delay, indicates the time the mouse pointer must remain in the same location after the fly over help has been displayed for the first time.

You can change the length of the first delay using the setInitialDelayTime member function or query what is currently set using the initialDelayTime member function. Use the setDelayTime member function to change the second delay and use delayTime to query what is currently set for that second delay.

You associate context-specific help for a window to a help message by specifying a window identifier. This identifier is used with an offset to load strings from a string table. Specify different offsets into the string table for the IFlyText and the ITextControl objects to display different help messages in each of the controls.

Note: To display help for a window using the string, you must either create the window using the User Interface Class Library or wrapper an existing window.

Dynamically Adding Help Text to Windows

You can also dynamically associate help text to a window using IFlyOverHelpHandler, a window handler. This is useful when you dynamically add controls to a canvas or push buttons to a tool bar.

Use the following functions to dynamically add or remove the help text specified for a window:

flyHelpText Returns the short help text for a window if you have dynamically added help text for the window.

longHelpText Returns the long help text for a window if you have dynamically added long help text for the window.

setHelpText Sets help text for a window by specifying a string or resouce ID. If you add help text to a window by the setHelpText functions, this text takes precedence over text that would otherwise be loaded from a string table.

removeHelpText Removes help text you added to a window through the setHelpText function.

For more information on these functions, see the *Open Class Library Reference* or refer to the tool bar examples in your samples directory. These examples show you how to incorporate fly over help into your applications.

Setting Time Intervals

A timer controls when certain events will occur based on a specified timed basis. You can use the ITimer class objects to set time-interval-based operations. The User Interface Class Library uses timers in fly over help (see “Adding Fly Over Help” on page 588) and with IAnimatedButton. See the *Open Class Library Reference* for more information on the IAnimatedButton class. The User Interface Class Library provides the following classes to create timer objects:

ITimer This class creates objects representing operating system timers. An object of this class calls a specified function in your class when the timer expires. The timer continues to call the function each time the specified time limit expires.

Time Intervals

ITimerFn An abstract timer function class. An instance of this class is passed into ITimer when you start it. You can then delete the ITimer object, and the timer will continue to run.

ITimerMemberFn Template class for the timer member function. This class inherits from ITimerFn.

ITimerMemberFn0 Another template class for the timer member function. This class does not accept any parameters.

Creating Timers

Use the ITimer class objects to define and set time-interval-based operations for your current program. You can also use ITimer objects to create additional time-interval-based operations by instantiating new ITimer objects and starting them. You start a timer using the ITimer::start member function.

The following is an example of starting a new timer to execute a member function:

```
ITimer timer
timer.start( new ITimerMemberFn< MyClass >( *this, foo ) );
```

You construct instances of this class in one of the following ways:

- Using the default constructor. Use this to build an ITimer that is started using the ITimer::start member function.
- With specification of code to run. This form of the constructor is used to construct a new ITimer and immediately start it. It is equivalent to constructing using the default constructor and then using ITimer::start. You can specify a time-interval to be used (in thousandths of a second) with this constructor. A time-interval of one second is the default.
- From the timer ID of a currently started ITimer object.

Note: ITimer has a virtual destructor that deallocates resources. This destructor, however, does not stop the timer. The timer continues to run until you call ITimer::stop.

Using the Abstract and Template Classes

The ITimerFn class is an abstract timer function class. Objects of this class represent events dispatched when an active timer expires. These objects are reference counted to manage their destruction after the timer has stopped. This reference to an ITimerFn is then passed to ITimer::start. The timerExpired function is called when the timer expires.

The ITimerMemberFn template class is an ITimerFn-derived class for dispatching C++ member functions against an object when a timer expires.

Time Intervals

You can use instances of this class to dispatch C++ member functions when a timer expires. The template argument is the class of the object for which the member functions are called. The constructor for such objects requires a reference to the object the member function is to be applied to and a pointer to the member function.

This member function must return a void and accept a timer reference argument.

Note: This class overrides the inherited function, timerExpired.

To create a timer you would do the following:

1. Write a function for what you want to do on a timed basis.
2. Create a ITimerMemberFn or ITimerMemberFn0 object. For a function that will be repeating an action on a timed basis, you would use ITimerMemberFn0. An example would be if you wanted a window to refresh itself every 30 minutes. If you wanted a window displayed for five seconds and then disappear, you would use ITimerMemberFn since the action only occurs once. You would then use ITimer::stop to delete the timer.

The following is an example of a timer created to run the public member job in MyClass:

```
class MyClass {
public:
void job ( unsigned long timerId )
  {
  // Code to run.
  }
};
//...
MyClass myObj;
ITimer  timer;
timer.start( new ITimerMemberFn<MyClass>( myObj, MyClass::job ) );
```

In the preceding example, we used ITimerMemberFn because the job member function returns void and accepts an unsigned long parameter, timerId. When parameters are not accepted by the member function, you can use ITimerMemberFn0.

See the *Open Class Library Reference* for more information on the timer classes.

Time Intervals

45 Introducing the Sample Application

Sample application files are provided with the User Interface Class Library product diskettes. Use the samples to understand the classes. Complete listings are included in the `\ibmcpp\samples\ioc` directory.

About the Hello World Application

The Hello World sample application is divided into several versions, starting with the simplest form, version 1, and building up to the most complicated form, version 6. Each version shows you a different aspect of the User Interface Class Library.

Chapter 46, "Creating a Main Window" through Chapter 51, "Adding a Font Dialog, a Pop-up Menu, and a Notebook" show you how to build an application, called "Hello World", using the User Interface Class Library. This sample application does not teach you C++ programming. If you are not familiar with the principles and aspects of C++ programming, consult the *C++ Language Reference* before continuing with this section.

Running the Hello World Files

Files are included to help you compile and link each version of the Hello World sample application. The README file contains complete instructions for compiling and linking each version.

Reviewing the Conventions Used in the Samples

The User Interface Class Library uses conventions to enhance the usability and readability of the code. The following conventions will help you as you create applications.

- Class names begin with a capital letter. For example, all classes belonging to the User Interface Class Library with a global scope begin with the letter "I", as in IApplication. If a class name consists of more than one word, the first letter of each word is capitalized, such as IFrameWindow.

  In keeping with this standard, the letter "A" was chosen as the first letter (for example, AHelloWindow) for the Hello World application-defined classes. This convention helps you distinguish the Hello World application classes from the User Interface Class Library classes. This naming convention also helps you distinguish the classes you create from those supplied by the class library.

Hello World Conventions

- Member functions begin with a lowercase letter. If a member function name consists of more than one word, the first letter of each word that follows the first word is capitalized, such as setText.

  Note: In the *User's Guide*, single-word member functions have ClassName:: added to them, for example, the member function “show” appears as IWindow::show.

- A version indicator (for example, v2 or v4) appears in columns 79-80 in some sample code comments, indicating which statements were added to enhance the previous version. The following example illustrates this convention:

```
#include <istattxt.hpp>                   //IStaticText Class
#include <iinfoa.hpp>                     //IInfoArea Class
#include <imenubar.hpp>                   //IMenuBar Class
#include <ifont.hpp>                      //IFont
#include <istring.hpp>                    //IString Class
#include <isetcv.hpp>                     //ISetCanvas Class
```

See “User Interface Class Library Conventions” on page 268 for information about other User Interface Class Library conventions.

46 Creating a Main Window

Version 1 of the Hello World sample application shows you how to create a main window and insert a text string into it using the static text control. A *static text control* is a text field, bit map, icon, or box that you can use to label or box another control. In version 1, the “Hello World!!!” text string is inserted into a static text control.

Version 1 shows you how to do the following:

1. Create the main window
2. Create a static text control
3. Set the focus and show the main window

The main window for version 1 of the application looks like this:

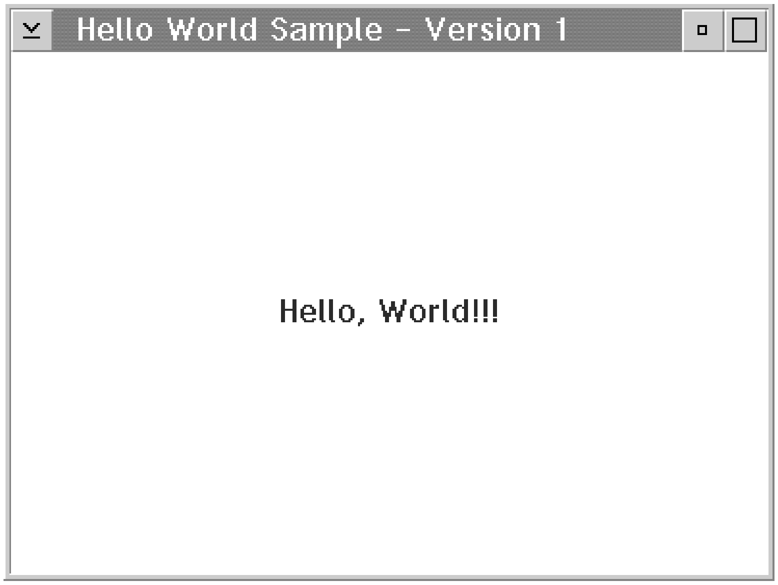


Figure 82. Version 1 of the Hello World Sample Application

Listing the Hello World Version 1 Files

The AHELLOW1.CPP file contains the source code for the main procedure, which is described later in this chapter.

Hello World Version 1

File	Type of Code
AHELLOW1.CPP	Source code for the main procedure

The Primary Source Code File

AHELLOW1.CPP contains the source code used for version 1. Here is a listing of the source code:

```
⋮
//Include User Interface Class Library class headers:
#ifndef _IBASE_                        //Make sure ibase.hpp is included
  #include <ibase.hpp>                 //  since that is where IC_<environ>
#endif                                 //  is defined.
#include <iapp.hpp>                    //IApplication class
#include <istattxt.hpp>                //IStaticText class
#include <iframe.hpp>                  //IFrameWindow class

/**************************************************************************
* main  - Application entry point for Hello World Version 1.             *
*         This simple application does the following:                     *
*           1) Creates a new object mainWindow of class IFrameWindow      *
*           2) Creates a new object hello of class IStaticText            *
*           3) Sets the static text value and aligns it                   *
*           4) Sets the static text as the client of the mainWindow       *
*           5) Sets the size of mainWindow                                *
*           6) Sets the window focus to mainWindow                        *
*           7) Displays the mainWindow                                    *
*           8) Starts the events processing for the application           *
**************************************************************************/
int main()
{
  IFrameWindow mainWindow ("Hello World Sample - Version 1", 0x1000);

  IStaticText hello(0x8008, &mainWindow, &mainWindow);
  hello.setText("Hello, World!!!");
  hello.setAlignment(IStaticText::centerCenter);
  mainWindow.setClient(&hello);

  mainWindow.sizeTo(ISize(400,300));
  mainWindow.setFocus();
  mainWindow.show();
  IApplication::current().run();
  return 0;
} /* end main */
```

Exploring Hello World Version 1

The following sections describe each of the tasks performed by version the Hello World version 1 application.

Creating the Main Window

The first statement creates the main window, an instance of the IFrameWindow class, for the application. To make this class available, the application must include the IFRAME.HPP library header file, as follows:

```
⋮
#include <iframe.hpp>                 //IFrameWindow Class
⋮
```

Now that the IFrameWindow class is available, a variable, in this case mainWindow, is defined as a new object of this class. This object represents the main window of the application. For example:

```
⋮
  IFrameWindow mainWindow ("Hello World Sample - Version 1", 0x1000);
⋮
```

The window ID is assigned the hexadecimal value 0x1000. The object represents the main window of the application.

Creating a Static Text Control

Next, create a static text control for the “Hello, World!!!” text string. Because this control is an object of the IStaticText class, includes another library header file, ISTATTXT.HPP, as follows:

```
⋮
#include <istattxt.hpp>               //IStaticText class
⋮
```

Now, define another variable, hello, as a new object of the IStaticText class, which represents a static text control. Use the following code:

```
⋮
  IStaticText hello(0x8008, &mainWindow, &mainWindow);
⋮
```

The control ID is assigned the hexadecimal value 0x8008.

The argument that follows the hexadecimal value identifies the mainWindow as the parent of the static text control. This positions the static text control in relation to the main window and displays it on top of the main window.

The last argument identifies the main window as the owner of the static text control. Controls notify their owner windows when events take place by using command, help, or control events. In this case, if an action is performed on the static text control, such as modifying its text string, that action is reported to the main window, which is specified as the owner. In version 1, no actions are performed on the static text control, but they are in Versions 2 through 6.

Setting a Text String for the Static Text Control

After creating the static text control, give it a static text string. The IStaticText class is derived from the ITextControl class and, thus, inherits its member functions. One of those member functions, setText, defines the text string for the static text control. For example:

```
⋮
hello.setText("Hello, World!!!");
⋮
```

Aligning the Static Text Control

Next, the setAlignment member function of the IStaticText class aligns the text string in the static text control. In this sample, it is centered both horizontally and vertically.

```
⋮
hello.setAlignment(IStaticText::centerCenter);
⋮
```

If you do not align the text string, the default placement is in the upper left corner of the static text area.

Setting Static Text Control as the Client Window

Next, designate the static text control as the frame's client window so that the "Hello, World!!!" text string displays in the main window's client area. Use the setClient member function of the IFrameWindow class, as follows:

```
⋮
mainWindow.setClient(&hello);
⋮
```

The frame's client window is the window corresponding to the client area, which is the rectangular portion of the frame window not occupied by the other frame controls (for example, title bar, window border, and minimize and maximize buttons). Setting the static text control as the client window causes it to occupy the entire client area and to be aligned within the boundaries of that area. When the user resizes the main window, the client area (static text control in this sample) grows or shrinks.

Setting the Size of the Main Window

The following code shows you how to change the size of the main window:

```
⋮
mainWindow.sizeTo(ISize(400,300));
⋮
```

This sets the size of the main window to 400 pixels wide by 300 pixels high.

Setting the Focus and Showing the Main Window

The two statements in the following code do the following:

- Designate the main window as the active window
- Display the main window when running the application

These statements use the IFrameWindow::setFocus and IWindow::show member functions:

```
⋮
mainWindow.setFocus();
mainWindow.show();
⋮
```

Because IFrameWindow is derived from IWindow, the setFocus and IWindow::show member functions are inherited from the IWindow class. Classes inherit functions from the base classes from which they are derived. An application does not have to include those base classes. Therefore, the IWindow class does not need to be included in this application for its functions to be available.

Starting Event Processing

The last statement displays the main window and starts the user interface window event processing for the application. This is accomplished by using member functions belonging to the IApplication and ICurrentApplication classes. Therefore, include another library header file, IAPP.HPP, as follows:

```
⋮
#include <iapp.hpp>                   //IApplication class
⋮
```

The IApplication::current member function of the IApplication class returns the current application as an instance of the ICurrentApplication class. Next, the ICurrentApplication::run member function displays the main window and starts event processing for this application, using the following code:

```
⋮
IApplication::current().run();
⋮
```

Hello World Version 1

47 Adding a Resource File and Frame Extensions

Version 2 of the Hello World application shows you how to use a resource file and how to add frame extensions to the application window.

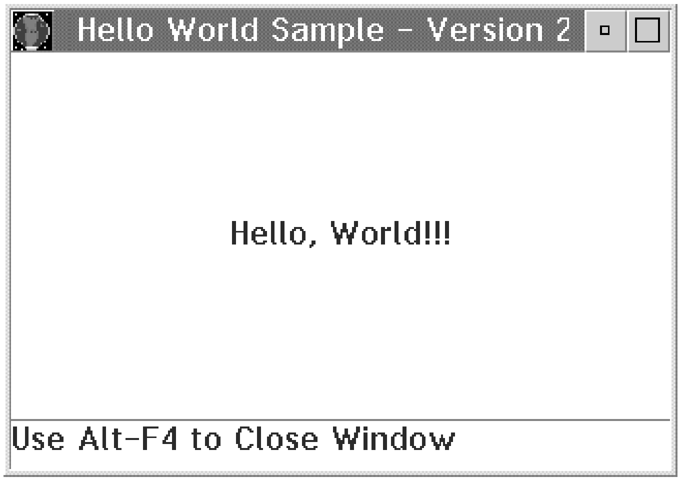
A *resource file* is a file that contains data used by an application, such as text strings and icons. This data is often easier to maintain in a resource file than in the source code of an application because the resource file keeps all of the application's data together in one place.

Frame extensions are controls that you can add to a frame window in addition to those that are provided for you by basic Presentation Manager frame windows. For example, in version 2, an information area is added below the client area.

Version 2 of the Hello World application extends version 1 by showing you how to:

- Get the "Hello, World!!!" text string and text for an information area
- Construct the main window and set the title and system menu icon
- Create and set the information area below the client window

The main window for version 2 of the Hello World application looks like this:



Hello World Sample - Version 2

Hello, World!!!

Use Alt-F4 to Close Window

Figure 83. Version 2 of the Hello World Sample Application

Listing the Hello World Version 2 Files

The following files contain the code used to create version 2:

File	Type of Code
AHELLOW2.CPP	Source code for the main procedure and window constructor
AHELLOW2.HPP	Header file for the AHellowWindow class
AHELLOW2.H	Symbolic definitions file for HELLO2.EXE
AHELLOW2.RC	Resource file for HELLO2.EXE
AHELLOW2.ICO	Icon file for HELLO2.EXE

The Primary Source Code File

The AHELLOW2.CPP file contains the source code for the main procedure and the window constructor. If columns 79-80 contain a //V, then this source line was modified or added in this version. The tasks performed by this code are described in the following sections.

The AHelloWindow Class Header File

The AHELLOW2.HPP contains the class definition and interface specifications for the AHelloWindow class, a subclass of IFrameWindow that was created specifically for this application. It is similar to a User Interface Class Library header file.

The Symbolic Definitions File

AHELLOW2.H contains the symbolic definitions for this application. These definitions provide the IDs for the application main window, controls, and text strings. They are required in this version of the Hello World application, because the text strings are pulled in from a resource file.

The Resource File

AHELLOW2.RC is the resource file, for version 2 of the Hello World application. This resource file assigns an icon and three text strings to the constants defined in the AHELLOW2.H file. AHELLOW2.H is included in this resource file so the icon and text strings can be associated with the appropriate IDs.

The User Interface Class Library for AIX supports the OS/2 Presentation Manager (PM) format for .RC files. This format provides a tag language for describing the following:

- String values
- Menu layout
- Accelerator key definitions
- Bit-map and icon associations to files

Note: OS/2 dialog templates are not supported.

The User Interface Class Library provides a tool, called ipmrc2X, for converting OS/2-style resources into AIX resources.

🕮 Refer to “Converting Resource Files” on page 463 for more information about the resource file conversion tool.

The Icon File

AHELLOW2.ICO is used as both the title bar icon and the icon that displays when the application is minimized.

The User Interface Class Library for AIX provides a tool, called ibmp2X, for converting OS/2 bit maps and icons into AIX .xpm files.

🕮 Refer to “Accessing Bitmap and Icon Resources” on page 460 for more information about the resource file conversion tool.

Here is how the icon appears when minimized:

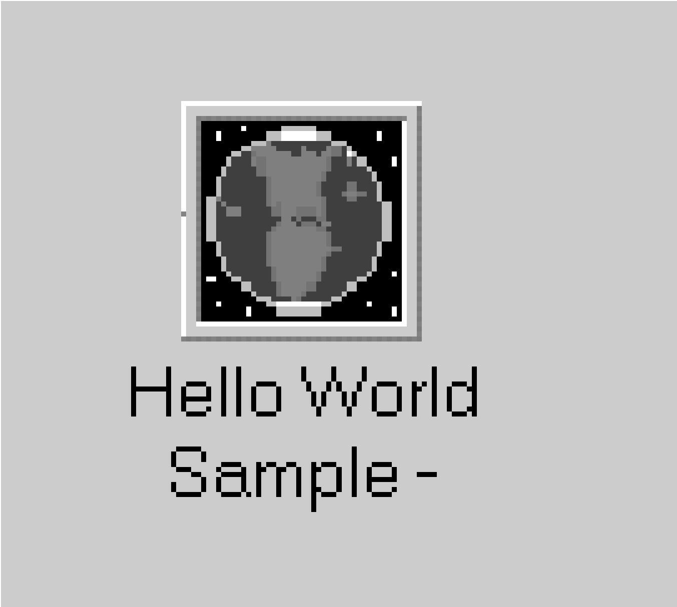


Figure 84. Hello World Icon

Discussing the Advantages of the C++ File Structure

In version 1, all of the source code was intentionally put in the AHELLOW1.CPP file to make that version of the application simple. However, for version 2, the source code has been distributed among a variety of files to show that you can structure your applications this way.

First, the AHelloWindow class, the subclass of IFrameWindow, is defined in the header file (AHELLOW2.HPP). Putting the class definition and interface specifications in the header file separates them from their implementation in the source code (AHELLOW2.CPP). This allows the class and its specifications to be used again with other applications and to be implemented in different ways. If the class definition or interface specifications change, for translation, for example, they change in only one place, the header file.

Similarly, the constant definitions file (AHELLOW2.H) assigns IDs to the windows and text strings in one place. Defining the constants this way allows you to use constants in a variety of places, such as the source code and the resource file, while keeping their definitions in one place. Then, if you need to change the constant definitions, you only modify the AHELLOW2.H file.

The advantage of placing the application's data in a resource file (AHELLOW2.RC) is that all of the resources are specified in one place. For example, finding and modifying text strings is easier when they are all grouped in one place, rather than searching through the source code for each one.

Exploring Hello World Version 2

The following sections describe each of the tasks performed by version 2 of the Hello World application. Some of the tasks are the same as those performed by version 1, but they are described again because they are performed differently in version 2.

Creating the Main Window

One of the major differences between version 1 and version 2 is the manner in which you create the main window. Version 1 simply creates an IFrameWindow object. However, version 2 provides its own class, AHelloWindow, to create the main window.

The AHelloWindow class is defined in the AHELLOW2.HPP header file and is derived from the IFrameWindow class. The IFrameWindow class is defined in the IFRAME.HPP library header file. Therefore, the AHELLOW2.HPP header file contains the following lines make the derivation of the AHelloWindow class from the IFrameWindow class possible:

```
⋮
#ifndef _IFRAME_
  #include <iframe.hpp>         //Include IFrameWindow class header
#endif
⋮
```

Hello World version 2 uses the compiler directive, ifndef, to prevent the IFRAME.HPP file from being included again, if it has already been included. This works, because by convention, the _IFRAME_ symbolic is defined in the IFRAME.HPP file. Both the User Interface Class Library and Hello World sample application use this convention in the header files.

See “Listing the Hello World Version 2 Files” on page 604 to learn about reasons for putting class definitions and interface specifications in a header file.

The AHELLOW2.CPP file, which contains most of the source code for the application, includes the AHELLOW2.HPP header file to have access to the AHelloWindow class:

```
⋮
#include "ahellow2.hpp"               //Include AHelloWindow class headers
⋮
```

The following lines in the AHELLOW2.CPP file create the main window by using the AHelloWindow class constructor:

```
⋮
AHelloWindow mainWindow (WND_MAIN);
⋮
```

In version 1, the main window is given a hexadecimal value of 0x1000 as its window ID when the main window was created. The same value is used for the window ID of the main window in version 2. However, instead of specifying that value in the primary source code file, Version 2 uses a constant, WND_MAIN, which is defined in the AHELLOW2.H file, as follows:

```
⋮
#define WND_MAIN         0x1000         //Main Window ID
⋮
```

Note: See “Listing the Hello World Version 2 Files” on page 604 to learn about reasons for using a constant definition file.

To have access to this definition, the primary source code file, AHELLOW2.CPP, must include the AHELLOW2.H file, as follows:

```
⋮
#include "ahellow2.h"                 //Include symbolic definitions
⋮
```

Starting Event Processing

When the main window is constructed, the following line in the AHELLOW2.CPP file gets the current application and runs it:

```
⋮
IApplication::current().run();
⋮
```

📖 See “Starting Event Processing” on page 601 for a more detailed explanation.

Constructing the AHelloWindow Object

Version 2 constructs the main window using the AHelloWindow class. Here is the class constructor as it is defined in the AHELLOW2.HPP header file:

```
⋮
class AHelloWindow : public IFrameWindow
{
public:
   AHelloWindow(unsigned long windowId);
⋮
```

In the primary source code file, AHELLOW2.CPP, version 2 uses the following lines to construct the main window:

```
⋮
AHelloWindow :: AHelloWindow(unsigned long windowId)
  : IFrameWindow(IFrameWindow::defaultStyle() |
                 IFrameWindow::minimizedIcon,
                 windowId)
   ,hello(WND_HELLO, this, this)
   ,infoArea(this)
{
⋮
```

Two capabilities provided by the IFrameWindow class used here were not used in version 1:

- Setting the main window to the default style

  Use the defaultStyle member function from the IFrameWindow class. It returns the current default style that all applications use for frame windows. The current default style is either the original default style that is provided by the User Interface Class Library for frame windows or a new default style that you establish by using the setDefaultStyle member function.

  In this case, because the setDefaultStyle member function has not been used, the current default style is the same as the original default style, which provides a title bar, title bar icon, minimize button, maximize button, window border, window list, and an initial shell position for the window.

🕮 Refer to “Adding Styles” on page 314 and to the *Open Class Library Reference* for more information about styles.

In this application, the title bar text and the application icon are specified in the resource file, AHELLOW2.RC. The text string for the window title is included in the resource file, and the icon, AHELLOW2.ICO, is specified.

- Displaying an icon when the main window is minimized

  The minimizedIcon style also inherits from the IFrameWindow class. This member function allows an application to use an icon to represent the application when it is minimized on the desktop. The Hello World application provides the AHELLOW2.ICO icon file for this purpose.

  🕮 Refer to Figure 84 on page 605 to see how this icon appears when the main window is minimized.

Creating a Static Text Control

Another difference between version 1 and version 2 is the means of creating a static text control to display a text string. In version 1, this was done simply by setting hello equal to a new instance of the IStaticText class, associating an ID with the control window (0x1010), and making the main window the parent and owner of the control.

In version 2, however, this code is divided into separate parts and placed in different files. As shown in the following lines, hello is now declared in the AHelloWindow class as an IStaticText object in the AHELLOW2.HPP file:

```
⋮
private:
  IStaticText   hello;
⋮
```

In the AHELLOW2.CPP file, hello points to a new instance of a static text control:

```
⋮
,hello(WND_HELLO, this, this)
⋮
```

The WND_HELLO constant provides the ID for the static text control. All Presentation Manager windows must have a unique ID, including controls. Therefore, the AHELLOW2.CPP file must include AHELLOW2.H, because that is where this constant is defined:

```
⋮
#include "ahellow2.h"                  //Include symbolic definitions
⋮
```

With the AHELLOW2.H included, the ID is associated with the WND_HELLO constant. The following code is from the AHELLOW2.H file:

```
⋮
#define WND_HELLO       0x8008        //Hello World Window ID
⋮
```

The other two arguments (this, this) pass in the main window (this instance of the AHelloWindow class) as the parent and owner of the static text control.

See “Creating a Static Text Control” on page 599 for information about parent and owner windows.

Setting Static Text Control as the Client Window

Next, set the static text control as the client window. The following code is from the AHELLOW2.CPP file:

```
⋮
setClient(&hello);
⋮
```

See “Setting Static Text Control as the Client Window” on page 600 for an explanation of client windows.

Setting a Text String for the Static Text Control

After the static text control is created, the next task is to set text in it. Version 2 gets the text string from a resource file. To do this, it uses the setText member function, which it inherits from the ITextControl class. The following code is in the AHELLOW2.CPP file:

```
⋮
hello.setText(STR_HELLO);
⋮
```

The setText member function finds this constant string in the AHELLOW2.RC file and puts it into the static text control:

```
⋮
STR_HELLO,  "Hello, World!!!"             //Hello World String
⋮
```

As noted earlier, each window, even a control, must have a numeric value assigned as its ID. The resource file includes the constant definition file, so this constant definition is available. The following code is from the AHELLOW2.H file.

```
⋮
#define STR_HELLO       0x1200        //Hello World String ID
⋮
```

Creating an Information Area

The following code, from the AHELLOW2.CPP file, creates a new instance of an information area using the IInfoArea class. This class provides a frame extension below the client window that shows information about the application.

```
⋮
,infoArea(this)
⋮
```

Setting the Information Area Text

Typically, the information shown in the information area pertains to the frame menu item at which the selection cursor is currently positioned. The information is loaded from a resource file string table. A different text string displays for each menu item, changing dynamically in the information area as the cursor moves from item to item. The information area also has a special string (called "inactive text") that displays whenever no menu item is selected.

Version 2 uses setInactiveText to set the information area's inactive text to the same string placed in the static text control in version 1. As a result, this text appears whenever the menu is inactive. The following code is from the AHELLOW2.CPP file:

```
⋮
infoArea.setInactiveText(STR_INFO);
⋮
```

The setInactiveText member function finds the STR_INFO constant in the AHELLOW2.RC file and puts it into the information area:

```
⋮
STR_INFO,   "Use Alt-F4 to Close Window"    //Information Area String
⋮
```

The STR_INFO constant is associated with a string ID, hexadecimal value 0x1220, in the AHELLOW2.H constant definition file. The resource file includes the constant definition file, so this constant definition is available.

```
⋮
#define STR_INFO        0x1220        //Info String ID
⋮
```

Aligning the Static Text Control

As in version 1, the static text control for the client area is centered both horizontally and vertically in the static text control. The following code is from the AHELLOW2.CPP file:

```
⋮
hello.setAlignment(IStaticText::centerCenter);
⋮
```

Hello World — Version 2

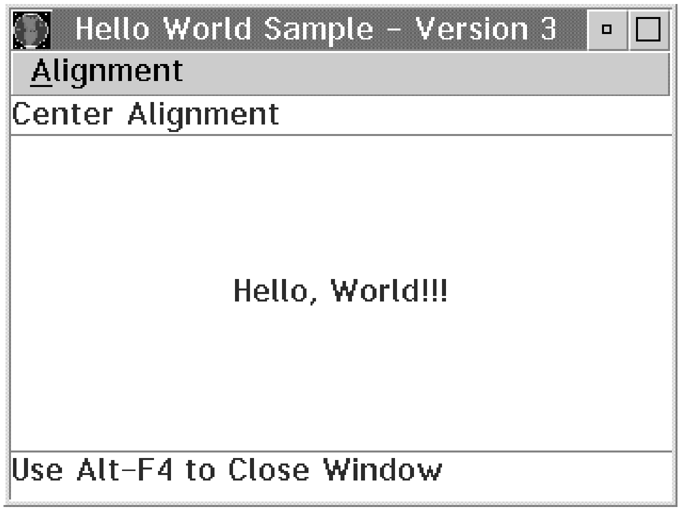
48 Adding a Command Handler and Menu Bars

Version 3 provides a menu bar with an **Alignment** choice. A *menu bar* is the area near the top of a window, below the title bar and above the client area of the window, which contains a list of choices. By selecting the **Alignment** choice, the user can display a pull-down menu and align the "Hello, World!!!" text string to the left, right, or center. In addition, this version adds a status area to show the status of the text string, and an event handler for the menu bar and the pull-down menu.

In covering these topics, this chapter shows you how to:

- Create a status line to show the status of the text string alignment
- Use an event handler
- Add a menu bar
- Set an initial check mark in the pull-down menu
- Add command processing (event handling) to align a text string

The main window for version 3 of the Hello World application looks like this:



Center Alignment

Hello, World!!!

Use Alt-F4 to Close Window

Figure 85. Version 3 of the Hello World Sample Application

Listing the Hello World Version 3 Files

The following files contain the code used to create version 3:

File	Type of Code
AHELLOW3.CPP	Source code for the main procedure, main window constructor, and command processing
AHELLOW3.HPP	Header file for the AHellowWindow and ACommandHandler classes
AHELLOW3.H	Symbolic definitions file for HELLO3.EXE
AHELLOW3.RC	Resource file for HELLO3.EXE
AHELLOW3.ICO	Icon file for HELLO3.EXE

The Primary Source Code File

The AHELLOW3.CPP file contains the source code for the main procedure and the AHelloWindow and ACommandHandler classes. The tasks performed by this code are described in the following sections.

The AHelloWindow Class Header File

AHELLOW3.HPP, like AHELLOW2.HPP, contains the class definitions and interface specifications for the AHelloWindow and ACommandHandler classes, with a few modifications for version 3.

The Symbolic Definitions File

AHELLOW3.H contains the definitions for this application. These definitions provide the IDs for the application window components.

For version 3, the symbolic definition file contains a new window ID (WND_STATUS) for the status area and three new string IDs (STR_CENTER, STR_LEFT, and STR_RIGHT) for the text strings used in the status area. In addition, menu IDs (MI_ALIGNMENT, MI_CENTER, MI_LEFT, and MI_RIGHT) have been added for the menu bar **Alignment** choice and the **Center**, **Left**, and **Right** choices in the pull-down menu.

The Resource File

Version 3 provides a resource file, AHELLOW3.RC. This resource file assigns an icon and several text strings with the constants defined in the AHELLOW3.H file. It also contains the text strings for the menu bar. AHELLOW3.H is included in this resource file so the icon and text strings can be associated with the appropriate IDs.

The resource file for version 3 contains two primary additions. The first is the text strings that are assigned to the new string constants that were defined in

AHELLOW3.H. These text strings are used in the status area to show the state of the static “Hello, World!!!” text string in the client area. For example, when the main window is first displayed, the “Center Alignment” text string is shown in the status area.

The second addition provides the text that appears on the menu bar (**Alignment**) and pull-down menu (**Left**, **Center**, and **Right**), indicating which choices are available. Each text string is assigned to a constant, also defined in AHELLOW3.H.

The tilde (˜) to the left of a letter in each text string indicates that the user can select those letters to select a menu item. These are shortcut keys for the application. For example, pressing R aligns the “Hello, World!!!” text string on the right side of the main window, just as if the **Right** choice had been selected from the pull-down menu.

The Icon File

AHELLOW3.ICO is used as both the title bar icon and the icon that displays when the application is minimized. This icon is the same as for version 2. Refer to Figure 84 on page 605 to see how this icon appears.

The User Interface Class Library for AIX provides a tool, called ibmp2X, for converting OS/2 bit maps and icons into AIX .xpm files.

📖 Refer to “Accessing Bitmap and Icon Resources” on page 460 for more information about the resource file.

Exploring Hello World Version 3

The following sections describe each of the tasks performed by version 3 of the Hello World application that have not been described for previous versions.

Constructing the AHelloWindow Object

Version 3 has made the following additions to the main window:

- Creating a status line
- Creating a menu bar
- Setting an initial check mark in the pull-down menu
- Aligning a text string
- Setting AHelloWindow as the event handler
- Destructing the main window

The following sections describe these additions.

Creating a Status Line

The status line shows the text string alignment status. Use the IStaticText class to create the static text control to display a text string in a status area. The *status area* is a small rectangular area that is usually located at the top of a window, below the menu bar.

In the AHELLOW3.CPP file, an IStaticText object, called statusLine, is created with this instance of the AHelloWindow class as the parent and owner.

```
⋮
,statusLine(WND_STATUS, this, this)
⋮
```

The WND_STATUS constant provides the window ID for this static text control. This constant is defined in AHELLOW3.H.

Specifying the Location and Height of the Status Area: Use the IFrameWindow member function addExtension in the AHELLOW3.CPP file to specify where the status area is positioned and how high it is. For example:

```
⋮
addExtension(&statusLine, IFrameWindow::aboveClient,
                IFont(&statusLine).maxCharHeight());
⋮
```

The aboveClient argument of the Location enumeration specifies that the static text control displays the status area above the client window.

The maxCharHeight member function returns the status area's maximum height, based on the current font.

Creating a Menu Bar

Now you can create the Alignment menu bar to display the **Left**, **Center**, and **Right** choices. In the header file, AHELLOW3.HPP, menuBar is defined as an instance of the IMenuBar class.

```
⋮
IMenuBar      menuBar;
⋮
```

AHELLOW3.CPP uses menuBar to create a new instance of that class in the main window, as follows:

```
⋮
,menuBar(windowId, this)
⋮
```

The WND_MAIN argument identifies the following menu in the AHELLOW3.RC resource file:

```
⋮
MENU WND_MAIN                                       //Main Window Menu (WND_MAIN)
  BEGIN
        SUBMENU "~Alignment", MI_ALIGNMENT      //Alignment Submenu
          BEGIN
            MENUITEM "~Left",       MI_LEFT     //Left Menu Item - F7 Key
            MENUITEM "~Center",     MI_CENTER   //Center Menu Item - F8 Key
            MENUITEM "~Right",      MI_RIGHT    //Right Menu Item - F9 Key
          END
  END
```

The window ID for the menu must match the window ID of the frame window.

This menu puts one choice, **Alignment**, on the menu bar, and provides a pull-down menu with three choices: **Left**, **Center**, and **Right**.

In addition, the MI_ALIGNMENT, MI_LEFT, MI_CENTER, and MI_RIGHT menu item attributes correspond to those in the resource file's string table:

```
⋮
MI_ALIGNMENT,"Alignment Menu"             //InfoArea - Alignment Menu
MI_CENTER,  "Set Center Alignment"        //InfoArea - Center Menu
MI_LEFT,    "Set Left Alignment"          //InfoArea - Left Menu
MI_RIGHT,   "Set Right Alignment"         //InfoArea - Right Menu
⋮
```

When the user moves the selection cursor over each menu item, the text string associated with that menu item displays in the information area below the client window. For example, when the cursor is on the **Right** menu item, the text string "Set Right Alignment" appears in the information area. For this to work, the string ID must match the corresponding menu item ID.

Setting an Initial Check Mark in the Pull-Down Menu

The pull-down menu that displays when the **Alignment** choice is selected on the menu bar contains three choices for aligning the "Hello, World!!!" text string: **Left**, **Center**, and **Right**. Because this text string is aligned in the center of the client area when the application is created, a check mark should display next to the **Center** choice the first time the pull-down menu displays.

The checkItem member function of the IMenuBar class lets you place a check mark on a pull-down menu choice. The following line, from in AHELLOW3.CPP, places a check mark on the **Center** choice:

```
⋮
menuBar.checkItem(MI_CENTER);
⋮
```

The MI_CENTER constant is defined in the AHELLOW3.RC resource file as the "Center" text string for the menu. Do not confuse this with the MI_CENTER menu item attribute defined in the string table, which is used only by the information area.

Destructing the AHelloWindow Object

After your application runs, you need to stop handling command events for the frame window and delete the objects you created using the new operator. The following lines in the AHELLOW3.CPP file to do this:

```
⋮
AHelloWindow :: ~AHelloWindow()
{
  commandHandler.stopHandlingEventsFor(this);
⋮
```

Aligning a Text String

This section shows you how to associate commands with the menu items to align the text string.

This sample shows command processing for one of the menu items. This code, from AHELLOW3.CPP, calls the AHelloWindow setAlignment function to center-align the "Hello, World!!!" text string in the client window:

```
⋮
IBase::Boolean
  ACommandHandler :: command(ICommandEvent & cmdEvent)
{
  Boolean eventProcessed(true);         //Assume event will be processed
⋮
  switch (cmdEvent.commandId()) {
    case MI_CENTER:
      frame->setAlignment(AHelloWindow::center);
      break;
⋮
```

The following code shows the setAlignment function from the AHelloWindow class.

```
AHelloWindow &
  AHelloWindow :: setAlignment(Alignment alignment)
{
⋮
  switch(alignment)
  {
⋮
  case center:
    hello.setAlignment(
      IStaticText::centerCenter);
    statusLine.setText(STR_CENTER);
    menuBar.checkItem(MI_CENTER);
    menuBar.uncheckItem(MI_LEFT);
    menuBar.uncheckItem(MI_RIGHT);
    break;
⋮
```

This code does the following:

- Uses the IStaticText setAlignment member function to center the static text control vertically and align it on the left horizontally
- Sets the appropriate text string in the status area (left alignment)
- Uses the uncheckItem member function to remove any existing check marks from the **Center** and **Right** menu items
- Uses the checkItem member function to set a check mark on the **Left**
- Returns true and ends

Adding Text for a Status Line

The status area text strings are specified in the resource file, as shown in the following code:

```
⋮
MI_CENTER,  "Set Center Alignment"          //InfoArea - Center Menu
MI_LEFT,    "Set Left Alignment"            //InfoArea - Left Menu
MI_RIGHT,   "Set Right Alignment"           //InfoArea - Right Menu
⋮
```

The following code, from AHELLOW3.CPP, gets the “Center Alignment” text string from the resource file and puts it in the static text control for the status area:

```
⋮
statusLine.setText(STR_CENTER);
⋮
```

Setting ACommandHandler as the Command Handler

In version 3, the AHelloWindow class contains a subclass of the ICommandHandler class, called ACommandHandler. This is necessary because, for the first time, this application handles events, in this case, the commands that align the “Hello, World!!!” text string.

The following code in AHELLOW3.CPP creates a command handler from the ACommandHandler class:

```
⋮
,commandHandler(this)
⋮
commandHandler.handleEventsFor(this);
⋮
```

The second line of code, shown above, contains the handleEventsFor member function inherited from the ICommandHandler class. Use this member function to set the event handler for the application. In this case, the this argument is specified.

This member function is available because the header file AHELLOW3.HPP includes the ICMDHDR.HPP library header file, which contains the ICommandHandler class.

```
⋮
#ifndef _ICMDHDR_
  #include <icmdhdr.hpp>         //Include ICommandEvent & ICommandHandler
#endif
⋮
```

Adding Command Processing

The next statements construct the command handler from a pointer to the AHelloWindow that events will be handled for. The AHELLOW3.CPP file uses the ACommandHandler to create a command handler, as follows:

```
⋮
ACommandHandler :: ACommandHandler(AHelloWindow *helloFrame)
{
   frame=helloFrame;                     //Save frame to be handled;
} /* end ACommandHandler :: ACommandHandler(...) */
⋮
```

Depending on the command event ID, you need to call the AHelloWindow::setAlignment function with the appropriate AHelloWorld::Alignment enumerator, as shown in the following sample from AHELLOW3.CPP. The AHelloWorld::Alignment enumerator is defined in the AHELLOW3.HPP file.

```
⋮
IBase::Boolean
  ACommandHandler :: command(ICommandEvent & cmdEvent)
{
  Boolean eventProcessed(true);         //Assume event will be processed
⋮
  switch (cmdEvent.commandId()) {
    case MI_CENTER:
      frame->setAlignment(AHelloWindow::center);
      break;
    case MI_LEFT:
      frame->setAlignment(AHelloWindow::left);
      break;
    case MI_RIGHT:
      frame->setAlignment(AHelloWindow::right);
      break;

    default:                            //Otherwise,
      eventProcessed=false;             //  the event wasn't processed
  } /* end switch */

  return(eventProcessed);
} /* end ACommandHandler :: command(...) */
⋮
```

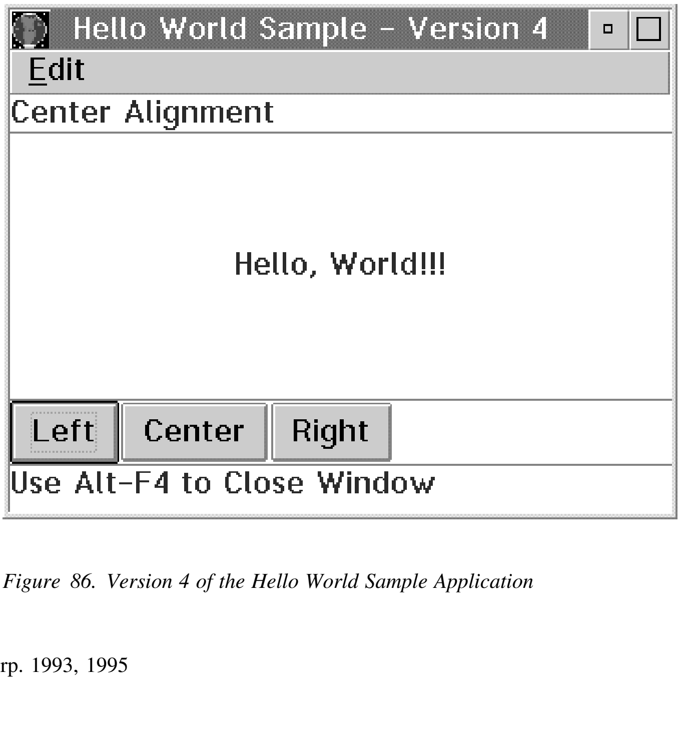
49 Adding Dialogs and Push Buttons

Version 4 modifies the menu bar and the pull-down menu in the following ways:

- Creates an **Edit** choice on the menu bar
- Moves the **Alignment** choice from the menu bar to the pull-down menu
- Moves the menu items associated with the **Alignment** choice (**Left**, **Center**, and **Right**) from the pull-down menu into a cascaded menu that displays when the **Alignment** choice is selected. These items still align the "Hello, World!!!" text string in the client window. However, the commands assigned to these menu items are also assigned to accelerator keys so the keyboard can bypass the menu choices and establish the text alignment.
- Adds a **Text...** choice on the pull-down menu. Selecting this choice displays a dialog box that contains an entry field in which the "Hello, World!!!" text string can be edited.

Hello World version 4 application contains a resource file from the User Interface Class Library for OS/2 Version 2.01, with definitions for accelerators. In OS/2, the accelerator definitions require that you specify `#include <os2.h>` in the .RC file.

The main window for version 4 of the Hello World application looks like this:



Center Alignment

Hello, World!!!

Use Alt-F4 to Close Window

Figure 86. Version 4 of the Hello World Sample Application

Listing the Hello World Version 4 Files

The following files contain the code used to create version 4:

File	Type of Code
AHELLOW4.CPP	Source code for the main procedure, main window constructor, and command processing
AHELLOW4.HPP	Header file for the AHellowWindow class
AHELLOW4.H	Symbolic definitions file for HELLO4.EXE
ADIALOG4.CPP	Source code to create the ATextDialog class
ADIALOG4.HPP	Header file for the ATextDialog class
AHELLOW4.RC	Resource file for HELLO4.EXE
AHELLOW4.ICO	Icon file for HELLO4.EXE

The Primary Source Code File

The AHELLOW4.CPP file contains the source code for the main procedure and the AHelloWindow and ACommandHandler classes. The tasks performed by this code are described in the following sections.

The AHelloWindow Class Header File

The AHELLOW4.HPP file, like the AHELLOW3.HPP file, contains the class definition and interface specifications for the AHelloWindow class, with a few modifications for version 4.

The Symbolic Definitions File

The AHELLOW4.H file contains the symbolic definitions for this application. These definitions provide the IDs for the application window components.

For version 4, the symbolic definition file contains the Hello World version 3 code, as well as the following:

- Three window IDs (WND_TEXTDIALOG, WND_MCCANVAS, WND_STCANVAS) for ATextDialog.
- One window ID for the push button set canvas (WND_BUTTONS). It also contains new string IDs (STR_CENTERB, STR_LEFTB, and STR_RIGHTB) for the text strings used in the push buttons.
- Two menu IDs (MI_EDIT and MI_TEXT) for the menu bar **Edit** choice and the **Text** choice in the pull-down menu.
- Four definitions for the dialog window controls (DID_OK, DID_CANCEL, DID_ENTRY, and DID_STATIC).

The Text Dialog Source Code File

The ADIALOG4.CPP file contains the source code for the ATextDialog class constructor and functions created for version 4.

The ATextDialog Class Header File

The ADIALOG4.HPP file contains the class definition and interface specifications for the ATextDialog class.

The Resource File

Version 4 provides a resource file, AHELLOW4.RC. This resource file assigns an icon and several text strings with the constants defined in the AHELLOW4.H file shown in “The Symbolic Definitions File” on page 622. It also contains resources for the menu bar and the accelerator keys.

AHELLOW4.H is included in this resource file so the icon, text strings, and other resources can be associated with the same IDs used in the application. OS2.H is included because it is the top level include file that includes all the files necessary for writing an OS/2 application.

The resource file for version 4 contains the version 3 code, as well as, additional strings, updated menus, and command IDs. The first is the accelerator table of command IDs assigned to the function keys. These command IDs are used in the cascaded menu to show the accelerator, or shortcut, key assignments. For example, with these assignments and the command processing in AHELLOW4.CPP, when users press the **F7** key, it is the same as if they select the **Left** choice in the cascaded menu.

📖 Refer to “Converting Resource Files” on page 463 for more information about the resource file conversion tool.

The Icon File

The AHELLOW4.ICO file is used as the icon that displays when the application is minimized. This icon is the same as for versions 2 and 3. Refer to Figure 84 on page 605 to see how this icon appears.

PM The User Interface Class Library for AIX provides a tool, called ibmp2X, for converting OS/2 bit maps and icons into AIX .xpm files.

📖 Refer to “Accessing Bitmap and Icon Resources” on page 460 for more information about the resource file.

Exploring Hello World Version 4

The following sections describe each of the tasks performed by version 4 of the Hello World application that have not been described for previous versions.

Adding a Cascaded Menu to the Menu Bar

For version 4, there are several modifications to the menu bar and its associated pull-down menu.

Version 4 replaces the **Alignment** menu bar choice with **Edit** and makes the **Alignment** choice a menu item on the **Edit** pull-down menu. When the **Edit** pull-down menu displays, an arrow to the right of the **Alignment** choice indicates that a cascaded menu will display to the right when it is selected. The **Alignment** and **Edit** choices are defined in the AHELLOW4.RC file, as follows:

```
⋮
MENU WND_MAIN                                //Main window menu bar
  BEGIN
    SUBMENU " Edit", MI_EDIT                 //Edit submenu
      BEGIN
        SUBMENU " Alignment", MI_ALIGNMENT      //Alignment submenu
          BEGIN
            MENUITEM " Left\tF7",   MI_LEFT     //Left menu item - F7 Key
            MENUITEM " Center\tF8", MI_CENTER   //Center menu item - F8 Key
            MENUITEM " Right\tF9",  MI_RIGHT    //Right menu item - F9 Key
          END
        MENUITEM " Text...", MI_TEXT           //Text dialog menu item
      END
  END
⋮
```

Adding Keyboard Accelerators

Keyboard accelerators are key sequences that perform the same actions as menu items. In version 3, the **Left**, **Center**, and **Right** choices appeared as items in a pull-down menu. In version 4, these choices become part of the cascaded menu and are assigned a function key. The menu items are defined in the AHELLOW4.RC file, with text describing the function and the accelerator key to use. The following code, from the AHELLOW4.RC file, shows the accelerator keys:

```
⋮
MENUITEM " Left\tF7",   MI_LEFT     //Left Menu Item - F7 Key
MENUITEM " Center\tF8", MI_CENTER   //Center Menu Item - F8 Key
MENUITEM " Right\tF9",  MI_RIGHT    //Right Menu Item - F9 Key
⋮
```

The `\t` indicates that the accelerator key name is tabbed to the right for readability.

This code conveys to users that the **Left**, **Center**, and **Right** alignment choices can be made by selecting a menu item with the mouse or keyboard, or they can use the keyboard function keys **F7**, **F8**, and **F9**.

The menu choices for **Left**, **Center**, and **Right** are visual indicators to the user that **F7**, **F8**, and **F9** can be used. The main window in AHELLOW4.CPP must be programmed to use the accelerator. This is done by using the accelerator style on the AHelloWindow constructor, as shown in the following code from the AHELLOW4.CPP file:

```
⋮
: IFrameWindow(IFrameWindow::defaultStyle() |
               IFrameWindow::minimizedIcon  |
               IFrameWindow::accelerator,
               windowId)
⋮
```

The default processing for this style causes the resource file to be searched for an ACCELTABLE definition for WND_MAIN which is the main window ID. The accelerator table for version 4 is defined in the AHELLOW4.RC file:

```
⋮
ACCELTABLE WND_MAIN                            //Acc. Table for Main Window
  BEGIN                                        //
    VK_F7,  MI_LEFT,   VIRTUALKEY              //F7 - Left Command
    VK_F8,  MI_CENTER, VIRTUALKEY              //F8 - Center Command
    VK_F9,  MI_RIGHT,  VIRTUALKEY              //F9 - Right Command
  END                                          //
⋮
```

Adding a Pull-Down Menu Choice

The final modification to the pull-down menu adds the **Text...** choice. By convention, the ellipsis (...) indicates that selecting this choice causes a dialog window to display. The following code from the AHELLOW4.RC file adds the **Text...** choice:

```
⋮
MENUITEM " Text...", MI_TEXT            //Text Menu Item
⋮
```

Figure 87 shows the pull-down menu choices and the cascaded menu.

Hello World Version 4

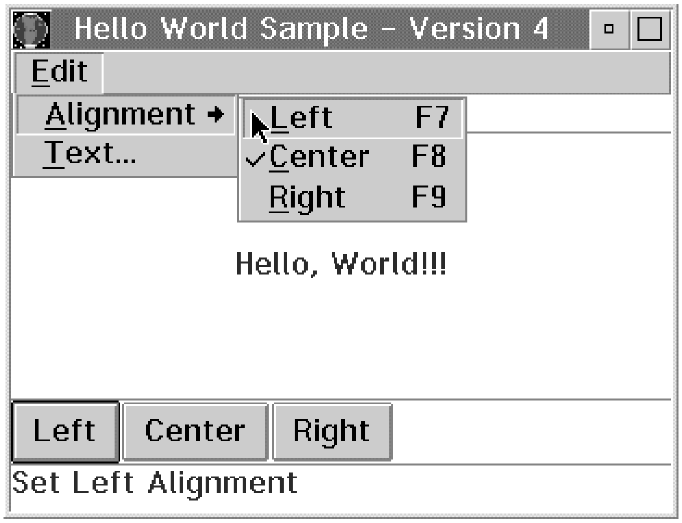


Figure 87. Cascaded Menu and Pull-Down Menu Choices for Version 4

Adding a Modal Dialog Window

A *dialog window* is a specific type of frame window containing window controls that gather information from the user. Typically, dialog windows are defined as modal to the owner frame window, that is, the user must respond to the dialog window before returning to the previous frame.

In OS/2 Presentation Manager (PM), you use dialog templates to define dialog windows externally to the application. The application creates a User Interface Class Library frame window with the same window ID defined in the resource file for the dialog template. Any controls from the dialog, for example entry fields, that you want to manipulate using the User Interface Class Library are also constructed from the corresponding control ID from the resource file. The following example shows an OS/2 dialog template that you could use with Hello World version 4.

```
DLGINCLUDE 1 "AHELLOW4.H"

DLGTEMPLATE WND_TEXTDIALOG LOADONCALL MOVEABLE DISCARDABLE
BEGIN
    DIALOG  "Hello World Edit Dialog", WND_TEXTDIALOG, 17, 22, 137, 84,
            WS_VISIBLE, FCF_SYSMENU | FCF_TITLEBAR
    BEGIN
        DEFPUSHBUTTON   "OK", DID_OK, 6, 4, 40, 14
        PUSHBUTTON      "Cancel", DID_CANCEL, 49, 4, 40, 14
        LTEXT           "Edit Text:", DID_STATIC, 8, 62, 69, 8
        ENTRYFIELD      "", DID_ENTRY, 8, 44, 114, 8, ES_MARGIN
    END
END
```

If you use the User Interface Class Library only for AIX or if you write portable applications, you should not use dialog templates, because they are not supported in AIX. Alternatively, you can use User Interface Class Library canvas controls.

📖 See Chapter 31, "Creating and Using Canvas Controls" on page 361 for information about the canvas classes. This Hello World version provides an example of using canvases.

In Hello World version 4, the dialog window contains the following:

- A prompt for the entry field. The prompt is a static text control with the text value loaded from the resource file.
- An entry field for changing the Hello World text string.
- An **OK** button, indicating the change should be returned to the main window.
- A **Cancel** button, indicating that the change should not be made.

Because these controls are best organized into columns and rows, use a multiple-cell canvas control. However, because all controls in a column must be the same width, aligning the **OK** button with the prompt would cause the **Cancel** button to be placed to the right of the prompt text. Therefore, place the push buttons in a set canvas and align the set canvas as a single control with the prompt and entry field.

The dialog looks like this:

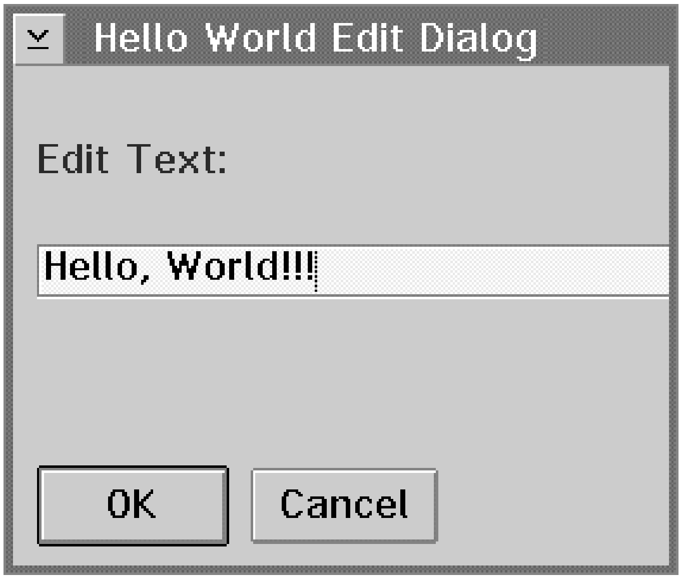


Figure 88. Dialog Window for Hello World Version 4

The following sections describe how to create, start, process, and end the dialog window.

Invoking the Dialog Window

As mentioned in the previous section, the **Text...** choice on the pull-down menu causes a dialog window to display. It does this by calling the AHelloWindow::editText member function, as shown in the following code from the AHELLOW4.CPP file, file.

```
⋮
    case MI_TEXT:
      frame->editText();
      break;
⋮
```

The editText function saves the hello text string and sets the information area to indicate that the text dialog is active. The following code shows this:

```
⋮
  IString textValue(hello.text());
  infoArea.setInactiveText(STR_INFODLG);
⋮
```

Version 4 creates the textDialog data member from the ATextDialog class, a new class created as a subclass of the IFrameWindow class. The following code comes from the AHELLOW4.CPP: file:

```
⋮
  ATextDialog textDialog(textValue,this);
⋮
```

The textValue object is passed as the current text string to the dialog.

Constructing the Dialog Window: When you create the textDialog object, the frame that represents the dialog and its components are initialized as shown in the following sample from the ADIALOG4.CPP file:

```
⋮
ATextDialog :: ATextDialog(IString & textString, IWindow * ownerWnd)
             : IFrameWindow(IResourceId(WND_TEXTDIALOG)
                  ,IWindow::desktopWindow()
                  ,ownerWnd
                  ,IRectangle(29,50,313,290)
                     .moveBy(ownerWnd->rect().bottomLeft())
                  ,IWindow::synchPaint
                    |IWindow::clipSiblings
                    |IWindow::saveBits
                    |dialogBackground
                    |dialogBorder
                    |systemMenu
                    |titleBar)
               ,clientCanvas(WND_MCCANVAS,this,this)
               ,buttons(WND_STCANVAS, &clientCanvas, &clientCanvas)
               ,statText(DID_STATIC,&clientCanvas,&clientCanvas)
               ,textField( DID_ENTRY,&clientCanvas,&clientCanvas)
               ,pushButton1( DID_OK,&buttons,&buttons)
               ,pushButton2(DID_CANCEL,&buttons,&buttons)
               ,dialogCommandHandler(this)
               ,saveText(textString)
{
⋮
```

The IFrameWindow is initialized, in this case, with values that correspond to the dialog template. The IRectangle values approximate the size and position that would be generated from the dialog specifications. The style bits correspond to the bits that would be set by the template. Notice that dialog windows typically do not have sizing borders or minimize and maximize buttons.

Next, initialize the six controls needed for the dialog. Note that the buttons are owned by the set canvas, that the set canvas, the static text, and the entry field are owned by the multiple-cell canvas; and that the multiple-cell canvas is owned by the dialog window frame.

Because there are buttons in the frame, you must initialize a command handler for handling the **OK** and **Cancel** buttons.

The saveText data member is initialized with the reference passed on the constructor. It provides the setTextFromEntryField access to the edit string.

Setting the Dialog Window Controls: Once you initialize the dialog window controls, you must position and set them. The following code, from the ADIALOG4.CPP file, positions and sets the dialog window controls:

```
⋮
textField.setText(saveText);
textField.disableAutoScroll().enableMargin().enableTabStop();

statText.setText(DID_STATIC);

pushButton1.enableDefault().setText(IResourceId(DID_OK));
pushButton2.setText(IResourceId(DID_CANCEL));
buttons.setPackType(ISetCanvas::expanded).setMargin(ISize());
buttons.enableTabStop();
⋮
```

First, you set the entry field. Then, you set the initial text value from the saved text value. Then set tabStop, autoScroll, and margin styles to match the settings within the dialog template.

You set the static text control's value from the resource file.

Set the push buttons by getting their text values from strings in the resource file and by enabling tabbing. Next, you set pushButton1 as the default push button, which means that when a user presses **Enter**, it is the same as pushing that button on the keyboard.

You only need to turn on padding and set margins to zero for the set canvas containing the buttons. Turning on padding lets the buttons expand to fill the set canvas area.

Next you position the controls in the multiple-cell canvas. Column 1 and rows 1-3, 5, 6, and 8-14 serve as padding to give the controls the correct spacing in the canvas. This is shown in the following code from the ADIALOG4.CPP file:

```
⋮
clientCanvas.addToCell(&statText , 2, 4);
clientCanvas.addToCell(&textField, 2, 7);
clientCanvas.addToCell(&buttons,   2,15);
⋮
```

Finally, you position the multiple-cell canvas in the dialog frame as the client, start the command handler for the dialog, and set the focus to the entry field. The following code, from the ADIALOG4.CPP file, shows this:

```
⋮
setClient( &clientCanvas );
dialogCommandHandler.handleEventsFor(this);
textField.setFocus();
⋮
```

Processing the Dialog Window: Once the textDialog has been created, editText displays and gives control to it using the AHelloWindow::showModally member function. Because the dialog window shows modally, it continues to have control until the dismiss function is called.

```
⋮
  textDialog.showModally();
⋮
```

Once you give the dialog window control by the AHelloWindow::showModally function, the user can interact with the dialog in three ways. Because you set the focus to the entry field, the user's normal keystrokes are processed by the default edit handler and the entry field is edited.

Secondly, the user can use the system menu to close or move the dialog window. However, the window cannot be resized.

Finally, the user can use the **Tab** keys or the mouse to select a push button. Also, because you defined the **OK** as a default key, users can use the **Enter** key to select it. When a user presses a button, the ADialogCommandHandler::command function processes the event. If the user presses **OK**, the setTextFromEntryField function is called to change the saved text value to the one edited in the text entry field. The code for this is found in the ADIALOG4.CPP file.

```
⋮
ATextDialog &
  ATextDialog::setTextFromEntryField()
{
  saveText = textField.text();
  return (*this);                      //Return a reference to the frame
} /* end AHelloWindow :: setTextFromEntryField */
⋮
```

The dismiss function is then called with the DID_OK ID. This value is saved in the textDialog's IFrameWindow object. The dismiss function closes the window and returns control to the owner window. This is found in the ADIALOG4.CPP file.

```
⋮
case DID_OK:
  frame->setTextFromEntryField();
  frame->dismiss(DID_OK);
  break;
⋮
```

Pressing the **Cancel** button does not call the setTextFromEntryField function, but it does call the dismiss function with the DID_CANCEL value. This is found in the ADIALOG4.CPP file, as follows:

```
⋮
case DID_CANCEL:
  frame->dismiss(DID_CANCEL);
  break;
⋮
```

The AHelloWindow::editText function can then use the textDialog.result function to determine if the user changed the text value. The information area is also reset. The following code is from the AHELLOW4.CPP file.

```
⋮
if (textDialog.result() == DID_OK)
      hello.setText(textValue);
infoArea.setInactiveText(STR_INFO);
⋮
```

The code for the text dialog comes from the ADIALOG4.CPP file. The declaration and interface specifications for the ATextDialog class are contained in the ADIALOG4.HPP file, which is included by both the AHELLOW4.CPP and ADIALOG4.CPP files.

Deleting the Dialog Window

Because you create the textDialog object statically, that is, the new operator is not used to create it, the object is deleted automatically when it is no longer in scope, in this case, when the editText function is exited. Using this approach means that each time the editText function is called, a new textDialog object is created, processed, and deleted.

Setting Push Buttons in a Set Canvas

The following code, from the AHELLOW4.HPP file, defines the buttons data member as an instance of the ISetCanvas class.

```
⋮
ISetCanvas    buttons;
⋮
```

📖 See “Understanding Set Canvases” on page 366 for more information about the ISetCanvas features described in this chapter.

To make the ISetCanvas class available to the application, the AHELLOW4.HPP file includes the ISETCV.HPP library header file, as follows:

```
⋮
#ifndef _ISETCV_
  #include <isetcv.hpp>         //Include ISetCanvas Class Header
#endif
⋮
```

Next, the buttons data member is created as a set canvas control with the main window as the parent and owner of the control. The WND_BUTTONS constant provides the window ID for this set canvas control. This is in AHELLOW4.HPP. AHELLOW4.H as follows:

```
⋮
#define WND_BUTTONS      0x1021         //Button Canvas Window ID
⋮
```

Use the setMargin and setPad member functions to set the canvas margins and pad to zero. The following code, from the AHELLOW4.CPP file, shows how to do this:

```
⋮
  buttons.setMargin(ISize());
  buttons.setPad(ISize());
  addExtension(&buttons, IFrameWindow::belowClient,
               (unsigned long)buttons.minimumSize().height());
⋮
```

Defining the Push Buttons

Now that you have a set canvas, define two push button data members in the header file, ADIALOG4.HPP, as shown in the following code:

```
⋮
IPushButton         pushButton1,
                    pushButton2;
⋮
```

Creating Push Buttons

The ADIALOG4.HPP, file includes the IPUSHBUT.HPP library header file and makes the IPushButton class available to version 4. You need the data members defined in the AHELLOW4.HPP file to create three push buttons in the set canvas: **Left**, **Center**, and **Right**. Use the following code to include the AHELLOW4.HPP file: AHELLOW4.CPP file uses the following code to include the IPUSHBUT.HPP file:

```
⋮
#ifndef _IPUSHBUT_
  #include <ipushbut.hpp>       //Include IPushButton Class Header
#endif
⋮
```

The following code creates a new instance of the **Left** push button control and specifies that it uses the command processing associated with the MI_LEFT menu item attribute to align the “Hello, World!!!” text string on the left side of the client window. The following code comes from AHELLOW4.CPP:

```
⋮
,leftButton(MI_LEFT, &buttons, &buttons)
⋮
```

Other than the data member used (centerButton is used for the **Center** push button and rightButton is used for the **Right** push button), the window ID is the only difference in the code that is used to create all three push buttons. Specify the MI_CENTER menu item window ID for the **Center** push button and MI_RIGHT for the **Right** push button.

The set canvas control is identified as the owner and parent of the push button control.

Setting Text in Push Buttons

The AHELLOW4.CPP file uses the setText member function to set text strings in each push button. Here is the code that sets the text in the **Left** push button:

```
⋮
leftButton.setText(STR_LEFTB);
⋮
```

Other than the data member for which the text is set (centerButton is used for the **Center** push button and rightButton is used for the **Right** push button), the only difference between this code and the code that puts text in the other two push buttons is the STR_LEFTB constant, which associates with the appropriate text string in the AHELLOW4.RC file. Here are the text string associations for all three push buttons:

```
⋮
STR_LEFTB,  "Left"                        //String for Left Button
STR_CENTERB,"Center"                      //String for Center Button
STR_RIGHTB, "Right"                       //String for Right Button
⋮
```

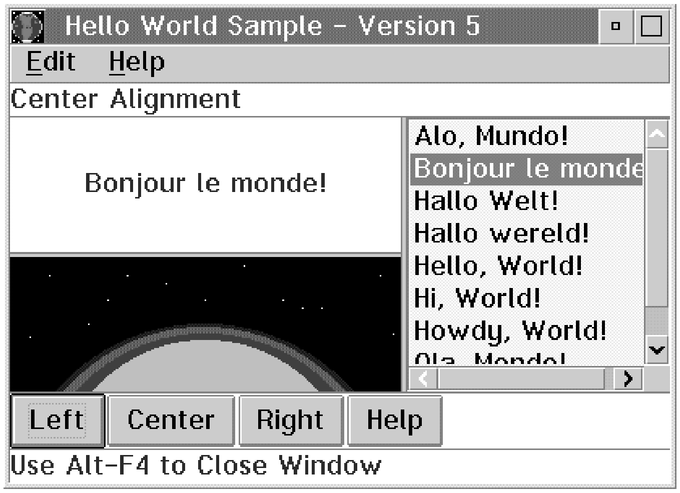
Hello World Version 4

50 Adding Split Canvases, a List Box, Native System Functions, and Help

Version 5 of the Hello World application shows you how to add the following:

- Split canvases for showing multiple control windows in the client window
- List box for selecting text for a static text window
- Native system functions to use in conjunction with User Interface Class Library objects
- Help window to provide users with information about the frame window

The main window for version 5 of the Hello World application looks like this:



Center Alignment

Bonjour le monde!

Alo, Mundo!
Bonjour le monde
Hallo Welt!
Hallo wereld!
Hello, World!
Hi, World!
Howdy, World!
Ola, Mondo!

Use Alt-F4 to Close Window

Figure 89. Version 5 of the Hello World Sample Application

Listing the Hello World Version 5 Files

The following files contain the code used to create version 5:

File	Type of Code
AHELLOW5.CPP	Source code for main procedure and AHelloWindow class
AHELLOW5.HPP	Class header file for AHelloWindow

Hello World Version 5

File	Type of Code
AHELLOW5.H	Symbolic definitions file for HELLO5.EXE
ADIALOG5.CPP	Source code for the ATextDialog class
ADIALOG5.HPP	Class header file for ATextDialog
AEARTHW5.CPP	Source code for the AEarthWindow class
AEARTHW5.HPP	Class header file for AEarthWindow
AHELLOW5.RC	Resource file for HELLO5.EXE
AHELLOW5.ICO	Icon file for HELLO5.EXE
AHELLOW5.IPF	Help source file for HELLO5.EXE

The Primary Source Code File

The AHELLOW5.CPP file contains the source code for the main procedure and AHelloWindow class functions. The tasks performed by this code are described in the following sections.

The AHelloWindow Class Header File

The AHELLOW5.HPP file contains the class definition and interface specifications for the AHelloWindow class, with a few modifications for version 5.

The Symbolic Definitions File

The AHELLOW5.H file contains the symbolic definitions for this application. These symbols and their definitions provide the IDs for the application window components.

The Text Dialog Source Code File

The ADIALOG5.CPP file contains the source code for the ATextDialog class, modified for help in version 5.

The ATextDialog Class Header File

The ADIALOG5.HPP file contains the class definition and interface specifications for the ATextDialog class. The ADIALOG5.HPP file is the same as the ADIALOG4.CPP file.

The Earth Window Source File

The AEARTHW5.CPP file contains the source code for the Earth window graphic that is drawn using native-system graphics calls.

The AEarthWindow Class Header File

The AEARTHW5.HPP file contains the class definition and interface specifications for the AEarthWindow class.

The Resource File

Version 5 of the Hello World application provides a resource file, AHELLOW5.RC, which contains all the resources from version 4, as well as additional resources, including a help table.

The Icon File

The AHELLOW5.ICO file is used as the icon that displays when the application is minimized. This icon is the same as for previous versions. Refer to Figure 84 on page 605 to see how this icon appears.

The Help Window Source File

The AHELLOW5.IPF file contains the text and IPF tags used to produce the help information for the Hello World application. IPF uses a tag language to format the text that appears in a help window. For example, :p. is the paragraph tag, which you use to start a new paragraph.

Refer to the *OS/2 Information Presentation Facility Guide and Reference* for descriptions of other tags used in the help source file. The makefiles provided with Hello World version 5 use the IPFC compiler, provided by the OS/2 Developer's Toolkit to compile the help file.

Exploring Hello World Version 5

The following sections describe each of the tasks performed by version 5 of the Hello World application that were not described for previous versions.

Constructing the Client Window with Split Canvases

In previous versions of the Hello World application, the client window contained a simple static text window. Version 5 provides an sample of a client window with three visible windows:

- A static text window containing “Hello World!!!” from the previous versions
- A new static text window with a graphical view of the earth from space
- A new list box containing different language versions of the phrase, “Hello World!!!”

The two new windows are described in detail in later sections of this chapter.

Use the ICanvas class for a common and easy-to-use method for placing multiple windows into a client window. Version 5 places the two static text windows into a horizontal split canvas, called helloCanvas, and then places helloCanvas and the list box into a vertical canvas called clientCanvas. It also places windows in a split canvas by identifying the canvas as the control window's parent. For example, the split canvases in the client area of the Hello World main window are initialized in the AHelloWindow constructor in the AHELLOW5.CPP file, as follows:

```
⋮
,clientWindow(WND_CANVAS, this, this)
,helloCanvas(WND_HCANVAS, &clientWindow, &clientWindow, IRectangle(),
            IWindow::visible | ISplitCanvas::horizontal)
,hello(WND_HELLO, &helloCanvas, &helloCanvas)
,earthWindow(WND_EARTH, &helloCanvas)
,listBox(WND_LISTBOX, &clientWindow, &clientWindow, IRectangle(),
            IListBox::defaultStyle() |
            IControl::tabStop |
            IListBox::noAdjustPosition)
⋮
```

The order in which you place your child windows into a split canvas determines the order that they will be seen. For example, the AHelloWindow::hello static text window appears above the earthWindow static text window because it is created as the first child of the helloCanvas window. Likewise, the helloCanvas window appears to the left of the list box because it is created before the list box.

After initializing the canvases, the following statements from the AHELLOW5.CPP file set the client window and the proportions for its child windows in the vertical split canvas.

```
⋮
setClient(&clientWindow);
clientWindow.setSplitWindowPercentage(&helloCanvas, 60);
clientWindow.setSplitWindowPercentage(&listBox, 40);
⋮
```

Creating and Using a List Box

Hello World version 5 provides you with the ability to change the Hello World text by selecting different language versions of the phrase “Hello World” from a list box. The IListBox object in Hello World version 5 is called listBox and is initialized in the AHelloWindow constructor in AHELLOW5.CPP as follows:

```
⋮
,listBox(WND_LISTBOX, &clientWindow, &clientWindow, IRectangle(),
            IListBox::defaultStyle() |
            IControl::tabStop |
            IListBox::noAdjustPosition)
⋮
```

The inherited tabStop style indicates that you can tab to the list box. The noAdjustPosition style prevents the list box from being automatically resized when an item in the list does not fit inside the current window.

One way you can populate the newly created list box is to use the IListBox member function, addAscending. This function adds a text string to the list box in ascending alphabetical order. For example, Hello World version 5 uses the addAscending function to load a variable number of strings from the resource file into the list box.

```
⋮
for (int i=0;i<HI_COUNT;i++ )
   listBox.addAscending(HI_WORLD+i);
⋮
```

The strings are defined in the resource file, AHELLOW5.RC, as follows:

```
⋮
// Change HI_COUNT in ahellow5.h to change number of HIs used.
HI_WORLD,      "Hello, World!"             //English
HI_WORLD+1,    "Hi, World!"                //American
HI_WORLD+2,    "Howdy, World!"             //Southern American
HI_WORLD+3,    "Alo, Mundo!"               //Portuguese
HI_WORLD+4,    "Ola, Mondo!"               //Spanish
HI_WORLD+5,    "Hallo wereld!"             //Dutch
HI_WORLD+6,    "Hallo Welt!"               //German
HI_WORLD+7,    "Bonjour le monde!"         //French
HI_WORLD+8,    "Put your language here!"   //Add more items, too!
⋮
```

HI_WORLD is a symbolic definition for the constant string ID of the first string. HI_COUNT is the symbolic definition for the constant number of strings to load. In this sample, HI_COUNT is defined as 8. Therefore, only HI_WORLD through HI_WORLD+7 are loaded. You can add another item to the list by changing the HI_WORLD constant in the symbolic definition file, AHELLOW5.H, and adding your strings to the HI_WORLD list.

Once you create the list box and fill it with items, define a select handler for processing list box selections. Hello World version 5 provides a select handler class, called ASelectHandler. (It is similar to the command handler added in Hello World version 3.)

The three differences are:

- The handler function overridden is called *selected*, instead of *command*
- The event passed into the function is an IControlEvent, instead of an ICommandEvent.
- The function being called processes selections made to our list box.

The ASelectHandler is defined in the AHelloWindow class header file, AHELLOW5.HPP, using the following class definition:

Hello World Version 5

```
⋮
class ASelectHandler : public ISelectHandler {
public:
  ASelectHandler(AHelloWindow *helloFrame);
protected:
virtual Boolean
  selected(IControlEvent& ctlEvent);
private:
  AHelloWindow *frame;
};
⋮
```

The ASelectHandler::selected function and the AHelloWindow::setTextFromListBox function provide the selection event handling for version 5. These functions are listed below and can be found in the AHELLOW5.CPP file.

```
⋮
IBase::Boolean
  ASelectHandler :: selected(IControlEvent & evt)
{
  frame->setTextFromListBox();
  return (true);                        //Event is always processed
} /* end ASelectHandler :: selected(...) */
⋮
AHelloWindow &
  AHelloWindow :: setTextFromListBox()
{
  IListBox::Cursor lbCursor(listBox);
  lbCursor.setToFirst();
  hello.setText(listBox.elementAt(lbCursor));
  return (*this);                       //Return a reference to the frame
}; /* end AHelloWindow :: setTextFromListBox() */
⋮
```

The setTextFromListBox function introduces a new User Interface Class Library class, IListBox::Cursor. Use a list box cursor to scan through the items in a list box. The constructor used in Hello World version 5 for creating the lbCursor object contains only one argument, the list box object to be scanned. You can also specify an additional argument, called a filter, to specify if you want to scan all the items in the list or only the items that are selected. Because the setTextFromListBox function looks for the first item selected, the default filter type, selected, is used.

The IListBox::Cursor::setToFirst function positions the cursor to the first selected item. Then, the IListBox::elementAt function uses the cursor to locate and return the text string identified by the first selected item. Hello World version 5 uses the string value to set the Hello World text.

Like the command handler in Hello World version 3, the ASelectHandler::selected function is not called until you attach the handler to the proper window. In this case, you attach the select handler to the list box using the ASelectHandler inherited

IHandler function, handleEventsFor. This causes the ASelectHandler::selected function to be called each time you select a list box item.

To stop handling selection events, use the stopHandlingEventsFor function, again specifying the list box.

Using Native System Functions and a Paint Handler

The AEARTHW5.HPP and AEARTHW5.CPP files contain the class header and implementation for a AEarthWindow class. Refer to the comments and code in these files for examples of the following:

- Calling native system functions from within user interface member functions
- Providing system-specific function in a portable application using compiler directives
- Displaying a graphics window using an IStaticText object
- Creating and using an IPaintHandler to repaint a static text window whenever it is invalidated
- Representing geometric shapes using IRectangle and IPoint objects
- Using IColor objects as arguments to native function calls

Setting Up the Help Area

Use the following steps to create help information for your application:

1. Create a file containing the help information.

   Create the source text that displays in your application's help window using the IPF format (.IPF file). Compile your IPF file into a help file (.HLP file) using the IPFC compiler.

   Refer to the *OS/2 Information Presentation Facility Guide and Reference* for descriptions of the tags you use to create the source .IPF file.

   For an example of an IPF source file, refer to the Hello World version 5 AHELLOW5.IPF file, which is described in Chapter 50, "Adding Split Canvases, a List Box, Native System Functions, and Help" on page 635.

2. Define the help window title and the help submenu in your resource file. In Hello World version 5, the help window title and help submenu are defined in the AHELLOW5.RC file, as follows:

   ```
   ⋮
   STR_HTITLE, "C++ Hello World - Help Window" //Help window title string
   ⋮
   SUBMENU "~Help", MI_HELP, MIS_HELP          //Help submenu
     BEGIN
       MENUITEM "~General help...",   SC_HELPEXTENDED, MIS_SYSCOMMAND
       MENUITEM "~Keys help...",      SC_HELPKEYS, MIS_SYSCOMMAND
       MENUITEM "Help ~index...",     SC_HELPINDEX, MIS_SYSCOMMAND
     END
   :
   ```

.

MI_HELP is the help menu ID.

Normally, you specify MIS_HELP for a menu item to cause a help event, rather than a command event, to be posted when the menu item is selected. OS/2 PM ignores MIS_HELP specified on submenu items.

When MIS_SYSCOMMAND is specified with the predefined SC_HELP* IDs, a system command event is generated. The default system command handler recognizes the predefined IDs and shows the appropriate help panel, except for SC_HELPKEYS, which by default does nothing. You can override this default processing for SC_HELPKEYS, using an IHelpHandler, which is described in a later step.

3. Define a help table in the resource file.

   The help table defines the relationship between the window ID and the general or contextual panel ID that is defined in the IPF file. The following help table is defined in the resource file, AHELLOW5.RC, for Hello World version 5:

```
HELPTABLE HELP_TABLE
  BEGIN
    HELPITEM WND_MAIN,       SUBTABLE_MAIN,   100
    HELPITEM WND_TEXTDIALOG,  SUBTABLE_DIALOG, 200
  END

HELPSUBTABLE SUBTABLE_MAIN                    //Main window help subtable
  BEGIN                                       //
    HELPSUBITEM WND_HELLO, 100                //Hello static text help ID
    HELPSUBITEM WND_LISTBOX,102               //List box help ID
    HELPSUBITEM MI_EDIT, 110                  //Edit menu item help ID
    HELPSUBITEM MI_ALIGNMENT, 111             //Alignment menu item help ID
    HELPSUBITEM MI_LEFT, 112                  //Left command help ID
    HELPSUBITEM MI_CENTER, 113                //Center command help ID
    HELPSUBITEM MI_RIGHT, 114                 //Right command help ID
    HELPSUBITEM MI_TEXT, 199                  //Text command help ID
  END                                         //

HELPSUBTABLE SUBTABLE_DIALOG                  //Text dialog help subtable
  BEGIN                                       //
    HELPSUBITEM DID_ENTRY, 201                //Entry field help ID
    HELPSUBITEM DID_OK, 202                   //OK command help ID
    HELPSUBITEM DID_CANCEL, 203               //Cancel command help ID
  END                                         //
```

   WND_HELLO and WND_LISTBOX are control IDs, MI_* are menu item IDs, and the DID_* are push button IDs. Each window ID is related to a help panel ID. In the preceding example, WND_MAIN and WND_HELLO both correspond to help panel ID 100. That is, pressing the **F1** key in the main window area displays the same help panel as selecting **General help...** from the **Help** submenu.

4. Create a help window object for your application window.

Use the IHelpWindow class to associate help information with an application window. Hello World version 5 defines the private data member, helpWindow, as an IHelpWindow object. It is initialized in the AHelloWindow constructor in AHELLOW5.CPP AHELLOW5.HPP.

```
⋮
class AHelpHandler : public IHelpHandler {
⋮
protected:
virtual Boolean
     keysHelpId(IEvent& evt);
};
⋮
```

5. Provide the overridden virtual function keysHelpId, which is called when keys help is requested. The following code, from the Hello World version 5 AHELLOW5.CPP file, shows how to implement this function.

```
⋮
IBase::Boolean AHelpHandler :: keysHelpId(IEvent& evt)
{
  evt.setResult(1000);                 //1000=keys help ID in
                                       //  ahellow5.ipf file

  return (true);                       //Event is always processed
} /* end AHelpHandler :: keysHelpId(...) */
⋮
```

   In the preceding code, the help panel ID for the Hello World version 5 keys help is set in the event result.

6. Start and stop help events processing.

   Your help handler, previously described, does not begin handling help events until the you use the handleEventsFor member function. For example, the following code causes the helpHandler to begin processing help events for this frame window:

```
⋮
helpHandler.handleEventsFor(this);
⋮
```

   Typically, you include this statement in the constructor for the frame window.

   KeyConcept.Note that the window which handles help events must be an associated window. That is, you should identify the window as the associated window on the IHelpWindow constructor or explicitly identify the window as an associated window using the IHelpWindow::setAssociatedWindow function.

   When you want to stop handling help events, for example, when you close your frame window, use the stopHandlingEventsFor member function, as follows:

```
⋮
helpHandler.stopHandlingEventsFor(this);
⋮
```

You typically include this statement in the destructor for the frame window.

7. Associate secondary frame windows with the parent window's help window.

   You can use an owner window's help window for secondary frame windows by using the IHelpWindow::setAssociatedWindow member function. This function adds the secondary window to the help event chain for a specific help window. Specify a pointer to the secondary window as the one argument to this function.

   In many cases, you will want to make this association in the constructor of the secondary frame window, but you will not be passed a pointer to the owner window's help window. To get a reference to the owner's help window, use the static IHelpWindow member function helpWindow, specifying the owner window as the argument.

   Hello World version 5 provides an example in the ADIALOG5.CPP file, as follows:

```
⋮
IHelpWindow::helpWindow(ownerWnd)->setAssociatedWindow(this);
⋮
```

8. Attach the following special handler to child frame windows in your application. This handler is needed so that help processes correctly for these windows.

```
class ChildFrameHelpHandler : public IHandler {
typedef IHandler Inherited;
/*****************************************************************************
*  This handler enables the OS/2 Help Manager to use help tables to display    *
*  contextual help for a child frame window (one whose parent window is not    *
*  the desktop).  This handler should only be attached to child frame windows. *
*****************************************************************************/
public:
virtual ChildFrameHelpHandler
 &handleEventsFor       ( IFrameWindow* frame ),
 &stopHandlingEventsFor ( IFrameWindow* frame );
protected:
virtual Boolean
  dispatchHandlerEvent  ( IEvent& evt );
ChildFrameHelpHandler
 &setActiveWindow       ( IEvent& evt, Boolean active = true );
private:
virtual IHandler
 &handleEventsFor       ( IWindow* window  ),
 &stopHandlingEventsFor ( IWindow* window  );
};

IBase::Boolean ChildFrameHelpHandler :: dispatchHandlerEvent ( IEvent& evt )
{
  switch ( evt.eventId() )
  {
     case WM_ACTIVATE:
        setActiveWindow(evt, evt.parameter1().number1());
        break;
     case WM_HELP:
        setActiveWindow(evt, true);
        break;
     default:
        break;
  } /* endswitch */

  return false;                    // Never stop processing of event
}

ChildFrameHelpHandler&
  ChildFrameHelpHandler :: setActiveWindow ( IEvent& evt,
                                             Boolean active )
{
  IHelpWindow* help = IHelpWindow::helpWindow(evt.window());
  if (help)
  {
     IFrameWindow* frame = 0;
     if (active)
     {
        frame = (IFrameWindow*)evt.window();
     }
     help->setActiveWindow(frame, frame);
  }
  return *this;
}

ChildFrameHelpHandler&
  ChildFrameHelpHandler :: handleEventsFor ( IFrameWindow* frame )
{
  IASSERTPARM(frame != 0);
  Inherited::handleEventsFor(frame);
  return *this;
}

ChildFrameHelpHandler&
  ChildFrameHelpHandler :: stopHandlingEventsFor ( IFrameWindow* frame )
{
  IASSERTPARM(frame != 0);
  Inherited::stopHandlingEventsFor(frame);
  return *this;
}

IHandler& ChildFrameHelpHandler :: handleEventsFor ( IWindow* window  )
{          // private to hide version in IHandler
  ITHROWLIBRARYERROR(IC_MEMBER_ACCESS_ERROR,
                     IErrorInfo::invalidRequest,
                     IException::recoverable);
  return *this;
}

IHandler& ChildFrameHelpHandler :: stopHandlingEventsFor ( IWindow* window  )
{          // private to hide version in IHandler
  ITHROWLIBRARYERROR(IC_MEMBER_ACCESS_ERROR,
                     IErrorInfo::invalidRequest,
                     IException::recoverable);
  return *this;
}
```

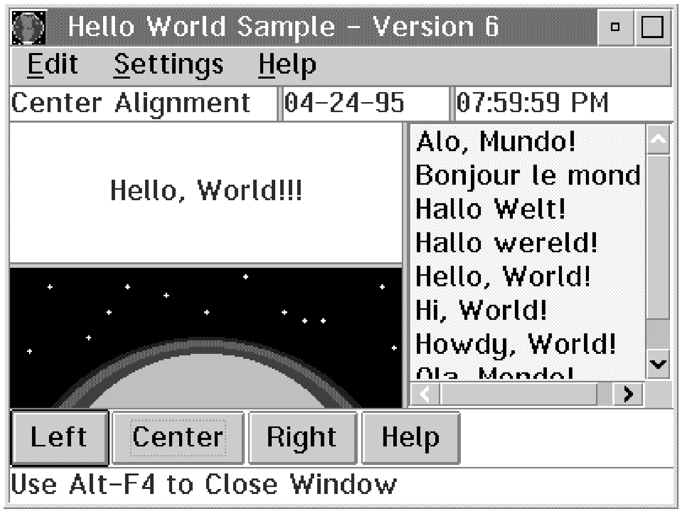
📖 Refer to the *Open Class Library Reference* for more information on IHelpWindow and IHelpHandler.

51 Adding a Font Dialog, a Pop-up Menu, and a Notebook

Version 6 of the Hello World application shows you how to do the following:

- Use a font dialog to change the font for a static text window
- Use a pop-up menu and a menu handler
- Use a notebook with multiple controls for changing application settings
- Use the IProfile class to save and read user settings
- Use a combination box instead of an entry field
- Add your own handler class for handling a type of event

The main window for Version 6 of the Hello World application looks like this:



Center Alignment 04-24-95 07:59:59 PM

Hello, World!!!

Alo, Mundo!
Bonjour le mond
Hallo Welt!
Hallo wereld!
Hello, World!
Hi, World!
Howdy, World!
Ola, Mondo!

Use Alt-F4 to Close Window

Figure 90. Version 6 of the Hello World Sample Application

Listing the Hello World Version 6 Files

The following files contain the code used to create version 6:

File	Type of Code
AHELLOW6.CPP	Source code for main procedure and AHelloWindow class
AHELLOW6.HPP	Class header file for AHellowWindow

Hello World Version 6

File	Type of Code
AHELLOW6.H	Symbolic definitions file for HELLO6.EXE
ADIALOG6.CPP	Source code for the ATextDialog class
ADIALOG6.HPP	Class header file for ATextDialog
AEARTHW6.CPP	Source code for the AEarthWindow class
AEARTHW6.HPP	Class header file for AEarthWindow
ANOTEBW6.CPP	Source code for the ANotebookWindow class
ANOTEBW6.HPP	Class header file for ANotebookWindow
ATIMEHDR.CPP	Source code for the ATimeHandler class
ATIMEHDR.HPP	Class header file for ATimeHandler
AHELLOW6.RC	Resource file for HELLO6.EXE
AHELLOW6.ICO	Icon file for HELLO6.EXE
AHELLOW6.IPF	Help file for HELLO6.EXE

Exploring Hello World Version 6

The following list describes the tasks performed by version 6 of the Hello World application that are not already described for previous versions. The tasks are as follows:

- Implementing a new time handler class derived from IHandler

  Refer to ATIMEHDR.CPP and ATIMEHDR.HPP for an example of writing your own handler. Hello World version 6 demonstrates writing your own handler by implementing a simple time handler that posts a WM_TIMER event and calls ATimeHandler::tick every second.

  Note: This class demonstrates IHandler derivation; the timer functions might not handle all cases and might break in a multithreaded environment.

  Refer to the AHELLOW6.CPP and AHELLOW6.HPP files for the AHelloTimeHandler class. The AHelloTimeHandler class overrides the ATimeHandler tick function to implement a running clock.

  You should also refer to the AEARTHW6.CPP and AEARTHW6.HPP files for ATwinkleTimeHandler. The ATwinkleTimeHandler overrides the ATimeHandler tick to provide stars that twinkle every second.

- Changing the status line in AHelloWindow to a split canvas with the status alignment plus the current date and time

Hello World version 6 also provides public functions for setting the date and time formats.

- Adding a pop-up menu and a menu handler to the AHelloWindow class

  Hello World version 6 derives a new class, APopUpHandler, from IMenuHandler, and overrides the virtual function makePopUpMenu to provide pop-up menus for the hello and earthWindow static text windows. This version also demonstrates using static and dynamic pop-up menus.

- Adding a new **Edit** submenu item, **Fonts...**

  This new menu item invokes the AHelloWindow::setHelloFont function that uses a modal IFontDialog to change the font in the AHelloWindow::hello static text window.

- Adding a new submenu to the **Settings** menu item

  This new submenu contains **Read**, **Open...**, and **Save** menu items.

- Adding a new Settings submenu item, **Open...**

  This new menu item invokes the AHelloWindow::openHelloSettings function that dynamically creates a nonmodal ANotebookWindow frame window. The nonmodal frame window lets you change AEarthWindow settings and date and time formats using check boxes, a slider, and radio buttons from a notebook window.

  Refer to ANOTEBW6.HPP and ANOTEBW6.CPP to see how the new ANotebookWindow class is defined and implemented, and to AHELLOW6.CPP for an example of its use.

- Adding a new **Settings** submenu item, **Save**

  This new menu item invokes the AHelloWindow::saveHelloSettings function that saves the hello font and the changeable user settings to an IProfile class file. Hello World version 6 uses a message box to indicate that the save completed successfully.

- Adding a new **Settings** submenu item, **Read**

  This new menu item invokes the AHelloWindow::readHelloSettings function that reads the hello font and the changeable user settings to an IProfile class file. Hello World version 6 uses a message box to indicate that the read completed successfully.

Hello World Version 6

- Enhancing the AEarthWindow function to include the following:
  - Setting and querying the intensity of the stars
  - Making the stars start and stop twinkling
  - Setting and querying the number of atmosphere layers
  - Setting and querying the color of the earth

  📖 Refer to the AEARTHW6.CPP and AEARTHW6.HPP files for more information.

Part 7. Appendices, Bibliography, Glossary, and Index

Class Hierarchy by Category

The following sections display the class hierarchies for each category of the User Interface Class Library

Application Control Classes

Provide support for the application, threads, timers, profiles, and the resources used by the applications you develop.

```
IBase
 ├─ICritSec
 ├─IHandle
 │  ├─IContextHandle
 │  ├─IProcessId
 │  ├─IProfileHandle
 │  ├─ISemaphoreHandle
 │  └─IThreadId
 ├─IProcedureAddress
 ├─IResourceId
 └─IVBase
    ├─IApplication
    │  └─ICurrentApplication
    ├─IClipboard
    ├─IClipboard::Cursor
    ├─IHandler
    │  └─IClipboardHandler
    ├─IProfile::Cursor
    ├─IProfile
    ├─IRefCounted
    │  ├─IThreadFn
    │  │  └─IThreadMemberFn
    │  └─ITimerFn
    │     ├─ITimerMemberFn
    │     └─ITimerMemberFn0
    ├─IResource
    │  ├─IPrivateResource
    │  └─ISharedResource
    ├─IResourceLibrary
    │  └─IDynamicLinkLibrary
    ├─IResourceLock
    ├─IThread
    │  └─ICurrentThread
    ├─IThread::Cursor
    ├─ITimer
    └─ITimer::Cursor
```

Base Window, Menu, Handler, and Event Classes

Provide support for the basic windows, handlers, events, and menus used by the applications you develop.

```
ISequence
 └─IFrameExtensions
IBase
 ├─IAccelerator
 ├─IBitFlag
 │  ├─IMenuItem::Attribute
 │  ├─IMenuDrawItemHandler::DrawFlag
 │  ├─IMenuItem::Style
 │  ├─IWindow::Style
 │  ├─IMessageBox::Style
 │  ├─IMenuBar::Style
 │  ├─IMenu::Style
 │  └─IFrameWindow::Style
 ├─ICoordinateSystem
 ├─IEventData
 ├─IEventParameter1
 ├─IEventParameter2
 ├─IEventResult
 ├─IFrameExtension
 ├─IHandle
 │  ├─IAccelTblHandle
 │  ├─IAnchorBlockHandle
 │  ├─IBitmapHandle
 │  │  └─ISystemBitmapHandle
 │  ├─IDisplayHandle
 │  ├─IEnumHandle
 │  ├─IMenuHandle
 │  ├─IMessageQueueHandle
 │  ├─IModuleHandle
 │  ├─IPointerHandle
 │  │  └─ISystemPointerHandle
 │  ├─IPresSpaceHandle
 │  ├─IStringHandle
 │  └─IWindowHandle
 ├─IHelpWindow::Settings
 ├─IHighEventParameter
 ├─ILowEventParameter
 ├─IMenuItem
 ├─ISWP
 ├─ISWPArray
 └─IVBase
    ├─IWindow::BidiSettings
    ├─IWindow::ChildCursor
    ├─ISubmenu::Cursor
    ├─IMenu::Cursor
    ├─IWindow::ExceptionFn
    └─IColor
       ├─IDeviceColor
       └─IGUIColor
```

Base Window, Menu, Handler, and Event Classes ...

```
IBase
 └─IVBase
    ├─IEvent
    │  ├─ICommandEvent
    │  ├─IControlEvent
    │  │  └─IDrawItemEvent
    │  │     └─IMenuDrawItemEvent
    │  ├─IFrameEvent
    │  │  └─IFrameFormatEvent
    │  ├─IHelpErrorEvent
    │  ├─IHelpHyperlinkEvent
    │  ├─IHelpMenuBarEvent
    │  ├─IHelpNotifyEvent
    │  ├─IHelpSubitemNotFoundEvent
    │  ├─IHelpTutorialEvent
    │  ├─IKeyboardEvent
    │  ├─IMenuEvent
    │  ├─IMouseEvent
    │  │  └─IMouseClickEvent
    │  ├─IMousePointerEvent
    │  ├─IPaintEvent
    │  └─IResizeEvent
    ├─IHandler
    │  ├─ICommandHandler
    │  ├─IEditHandler
    │  ├─IFocusHandler
    │  ├─IFrameHandler
    │  ├─IHelpHandler
    │  ├─IKeyboardHandler
    │  ├─IMenuDrawItemHandler
    │  ├─IMenuHandler
    │  ├─IMouseHandler
    │  │  └─IMousePointerHandler
    │  ├─IPaintHandler
    │  ├─IResizeHandler
    │  ├─ISelectHandler
    │  └─IWindowNotifyHandler
    │     ├─IFrameWindowNotifyHandler
    │     └─IMenuNotifyHandler
    ├─IMessageBox
    └─INotifier
       └─IWindow
          ├─IFrameWindow
          ├─IHelpWindow
          ├─IMenu
          │  ├─IMenuBar
          │  ├─IPopUpMenu
          │  ├─ISubmenu
          │  └─ISystemMenu
          └─IObjectWindow
```

Standard Control Classes

Provide support for the basic controls, such as entry fields, static text, and buttons used by the applications you develop.

```
IBase
 │ IBitFlag
 │  ├─I3StateCheckBox::Style
 │  ├─IBaseComboBox::Style
 │  ├─IBaseListBox::Style
 │  ├─IBaseSpinButton::Style
 │  ├─IButton::Style
 │  ├─ICheckBox::Style
 │  ├─IControl::Style
 │  ├─IEntryField::Style
 │  ├─IGroupBox::Style
 │  ├─IMultiLineEdit::Style
 │  ├─INumericSpinButton::Style
 │  ├─IOutlineBox::Style
 │  ├─IProgressIndicator::Style
 │  ├─IPushButton::Style
 │  ├─IRadioButton::Style
 │  ├─IScrollBar::Style
 │  ├─ISlider::Style
 │  ├─IStaticText::Style
 │  └─ITextSpinButton::Style
 └─IVBase
    ├─IBaseComboBox::Cursor
    ├─IBaseListBox::Cursor
    ├─ITextSpinButton::Cursor
    ├─IEvent
    │  ├─IControlEvent
    │  │  ├─IDrawItemEvent
    │  │  │  └─IListBoxDrawItemEvent
    │  │  └─IListBoxSizeItemEvent
    │  └─IScrollEvent
    └─IHandler
       ├─IListBoxDrawItemHandler
       ├─IScrollHandler
       ├─IShowListHandler
       ├─ISliderArmHandler
       ├─ISliderDrawHandler
       ├─ISpinHandler
       └─IWindowNotifyHandler
          ├─IListBoxNotifyHandler
          ├─IScrollBarNotifyHandler
          ├─INumericSpinButtonNotifyHandler
          ├─ITextSpinButtonNotifyHandler
          └─ITextControlNotifyHandler
             ├─IButtonNotifyHandler
             │  └─ISettingButtonNotifyHandler
             ├─IEntryFieldNotifyHandler
             │  └─IComboBoxNotifyHandler
             ├─IMultiLineEditNotifyHandler
             └─ITitleNotifyHandler
```

Standard Control Classes ...

```
IBase
 └─IVBase
    └─INotifier
       └─IWindow
          └─IControl
             ├─IBaseListBox
             │  └─IListBox
             ├─IBaseSpinButton
             │  ├─INumericSpinButton
             │  └─ITextSpinButton
             ├─IOutlineBox
             ├─IProgressIndicator
             │  └─ISlider
             ├─IScrollBar
             └─ITextControl
                ├─IButton
                │  ├─IPushButton
                │  └─ISettingButton
                │     ├─I3StateCheckBox
                │     ├─ICheckBox
                │     └─IRadioButton
                ├─IEntryField
                │  └─IBaseComboBox
                │     └─IComboBox
                ├─IGroupBox
                ├─IMultiLineEdit
                ├─IStaticText
                └─ITitle
```

Advanced Control, Dialog, and Handler Classes

Provide support for the advanced controls, such as container, notebook, and toolbar, and for the font and file dialogs used by the applications you develop.

```
ISequence
 └─ICnrObjectSet
ISet
 └─ICnrControlList
IBase
 ├─IBitFlag
 │  ├─IContainerControl::Attribute
 │  ├─INotebook::PageSettings::Attribute
 │  ├─IAnimatedButton::Style
 │  ├─IBitmapControl::Style
 │  ├─ICanvas::Style
 │  ├─ICircularSlider::Style
 │  ├─IContainerControl::Style
 │  ├─ICustomButton::Style
 │  ├─IDrawingCanvas::Style
 │  ├─IFileDialog::Style
 │  ├─IFontDialog::Style
 │  ├─IGraphicPushButton::Style
 │  ├─IIconControl::Style
 │  ├─IMultiCellCanvas::Style
 │  ├─INotebook::Style
 │  ├─ISetCanvas::Style
 │  ├─ISplitCanvas::Style
 │  ├─IToolBar::Style
 │  ├─IToolBarButton::Style
 │  ├─IToolBarContainer::Style
 │  └─IViewPort::Style
 ├─ICnrAllocator
 ├─IFileDialog::Settings
 ├─IFontDialog::Settings
 ├─IHandle
 │  └─IPageHandle
 └─IVBase
    ├─IContainerControl::ColumnCursor
    ├─IContainerControl::CompareFn
    ├─INotebook::Cursor
    ├─IContainerControl::FilterFn
    ├─IToolBar::FrameCursor
    ├─IContainerColumn
    └─IContainerObject
```

Advanced Control, Dialog, and Handler Classes ...

```
IBase
 └─IVBase
    ├─IEvent
    │  ├─ICnrDrawBackgroundEvent
    │  ├─IControlEvent
    │  │  ├─ICnrEvent
    │  │  │  ├─ICnrEditEvent
    │  │  │  │  ├─ICnrBeginEditEvent
    │  │  │  │  ├─ICnrEndEditEvent
    │  │  │  │  └─ICnrReallocStringEvent
    │  │  │  ├─ICnrEmphasisEvent
    │  │  │  ├─ICnrEnterEvent
    │  │  │  ├─ICnrHelpEvent
    │  │  │  ├─ICnrQueryDeltaEvent
    │  │  │  └─ICnrScrollEvent
    │  │  ├─ICustomButtonDrawEvent
    │  │  ├─IDrawItemEvent
    │  │  │  ├─ICnrDrawItemEvent
    │  │  │  └─INotebookDrawItemEvent
    │  │  └─IPageEvent
    │  │     ├─IPageHelpEvent
    │  │     ├─IPageRemoveEvent
    │  │     └─IPageSelectEvent
    │  └─IFileDialogEvent
    └─IHandler
       ├─ICnrDrawHandler
       ├─ICnrEditHandler
       ├─ICnrHandler
       ├─ICustomButtonDrawHandler
       ├─IFileDialogHandler
       ├─IFlyOverHelpHandler
       ├─IFontDialogHandler
       ├─IMenuHandler
       │  └─ICnrMenuHandler
       ├─IPageHandler
       ├─IShowListHandler
       ├─ISpinHandler
       └─IWindowNotifyHandler
          ├─INotebookNotifyHandler
          └─ITextControlNotifyHandler
             └─ICircularSliderNotifyHandler
```

Advanced Control, Dialog, and Handler Classes ...

```
IBase
 └─IVBase
    ├─INotifier
    │  └─IWindow
    │     ├─IControl
    │     │  ├─IBaseListBox
    │     │  │  └─ICollectionViewListBox
    │     │  ├─ICanvas
    │     │  │  ├─IDrawingCanvas
    │     │  │  ├─IMultiCellCanvas
    │     │  │  ├─ISetCanvas
    │     │  │  │  ├─IToolBar
    │     │  │  │  └─IToolBarContainer
    │     │  │  ├─ISplitCanvas
    │     │  │  └─IViewPort
    │     │  ├─IContainerControl
    │     │  ├─INotebook
    │     │  └─ITextControl
    │     │     ├─IButton
    │     │     │  ├─ICustomButton
    │     │     │  │  ├─IAnimatedButton
    │     │     │  │  └─IToolBarButton
    │     │     │  └─IPushButton
    │     │     │     └─IGraphicPushButton
    │     │     ├─ICircularSlider
    │     │     ├─IEntryField
    │     │        └─IBaseComboBox
    │     │               └─ICollectionViewComboBox
    │     │     ├─IFlyText
    │     │     └─IStaticText
    │     │        ├─IBitmapControl
    │     │        │  └─IIconControl
    │     │        └─IInfoArea
    │     └─IFrameWindow
    │        ├─IFileDialog
    │        ├─IFontDialog
    │        └─IToolBarFrameWindow
    ├─IRefCounted
    │  └─IStringGeneratorFn
    │     ├─IStringGeneratorMemberFn
    │     └─IStringGeneratorRefMemberFn
    ├─IContainerControl::Iterator
    ├─IContainerControl::ObjectCursor
    ├─INotebook::PageSettings
    ├─IContainerControl::TextCursor
    └─IToolBar::WindowCursor
```

Direct Manipulation Classes

Provide support for the direct manipulation used by the applications you develop.

```
IDM
IBase
 ├─IBitFlag
 │  └─IDMImage::Style
 └─IVBase
    ├─IDMImage
    ├─IDMItemProvider
    │  └─IDMItemProviderFor
    ├─IDMRenderer
    │  ├─IDMSourceRenderer
    │  └─IDMTargetRenderer
    ├─IEvent
    │  └─IDMEvent
    │     ├─IDMSourceBeginEvent
    │     ├─IDMSourceDiscardEvent
    │     ├─IDMSourceEndEvent
    │     ├─IDMSourcePrintEvent
    │     ├─IDMSourceRenderEvent
    │     │  └─IDMSourcePrepareEvent
    │     ├─IDMTargetEndEvent
    │     ├─IDMTargetEvent
    │     │  ├─IDMTargetDropEvent
    │     │  ├─IDMTargetEnterEvent
    │     │  └─IDMTargetLeaveEvent
    │     └─IDMTargetHelpEvent
    ├─IHandler
    │  └─IDMHandler
    │     ├─IDMSourceHandler
    │     └─IDMTargetHandler
    └─IRefCounted
       ├─IDMItem
       │  ├─IDMCnrItem
       │  ├─IDMEFItem
       │  ├─IDMMenuItem
       │  ├─IDMMLEItem
       │  ├─IDMTBarButtonItem
       │  └─IDMToolBarItem
       └─IDMOperation
          ├─IDMSourceOperation
          └─IDMTargetOperation
```

2D Graphic Classes

Provide support for the 2D graphic elements used by the applications you develop.

```
IBase
 ├─IGraphicBundle
 ├─IHandle
 │  └─IRegionHandle
 ├─ITransformMatrix
 └─IVBase
    ├─IGList::Cursor
    ├─IFont::FaceNameCursor
    ├─IFont
    ├─IGraphic
    │  ├─IG3PointArc
    │  ├─IGArc
    │  ├─IGBitmap
    │  ├─IGEllipse
    │  ├─IGLine
    │  ├─IGList
    │  ├─IGPie
    │  │  └─IGChord
    │  ├─IGPolyline
    │  │  └─IGPolygon
    │  ├─IGRectangle
    │  ├─IGRegion
    │  └─IGString
    ├─IGraphicContext
    └─IFont::PointSizeCursor
```

Dynamic Data Exchange Classes

Provide support for the Dynamic Data Exchange (DDE) used by the applications you develop.

```
ISet
 ├─IDDEActiveServerSet
 └─IDDEClientHotLinkSet
IBase
 ├─IDDE
 ├─IDDEActiveServer
 └─IVBase
    ├─IEvent
    │  ├─IDDEBeginEvent
    │  ├─IDDEEndEvent
    │  │  └─IDDEClientEndEvent
    │  └─IDDEEvent
    │     ├─IDDEAcknowledgeEvent
    │     │  ├─IDDEAcknowledgeExecuteEvent
    │     │  ├─IDDEAcknowledgePokeEvent
    │     │  ├─IDDEClientAcknowledgeEvent
    │     │  └─IDDEServerAcknowledgeEvent
    │     └─IDDESetAcknowledgeInfoEvent
    │        ├─IDDEClientHotLinkEvent
    │        ├─IDDEDataEvent
    │        ├─IDDEExecuteEvent
    │        ├─IDDEPokeEvent
    │        ├─IDDERequestDataEvent
    │        └─IDDEServerHotLinkEvent
    └─IHandler
       ├─IDDEClientConversation
       └─IDDETopicServer
```

Multimedia Classes

Provide support for the multimedia devices and controls used by the applications you develop.

```
IBase
 ├─IMMAudioBuffer
 └─IVBase
    ├─IMMAudioCDContents::Cursor
    ├─IErrorInfo
    │  └─IMMErrorInfo
    ├─IEvent
    │  ├─IMMCuePointEvent
    │  ├─IMMDeviceEvent
    │  ├─IMMNotifyEvent
    │  ├─IMMPassDeviceEvent
    │  └─IMMPositionChangeEvent
    ├─IHandler
    │  ├─ICommandHandler
    │  │  └─IMMPlayerPanelHandler
    │  └─IMMDeviceHandler
    │     └─IMMRemovableMediaHandler
    ├─IMMAudioCDContents
    ├─IMMSpeed
    ├─IMMTime
    │  ├─IMMHourMinSecFrameTime
    │  │  ├─IMM24FramesPerSecondTime
    │  │  ├─IMM25FramesPerSecondTime
    │  │  └─IMM30FramesPerSecondTime
    │  ├─IMMHourMinSecTime
    │  ├─IMMMillisecondTime
    │  ├─IMMMinSecFrameTime
    │  └─IMMTrackMinSecFrameTime
    └─INotifier
       ├─IStandardNotifier
       │  ├─IMMDevice
       │  │  ├─IMMAmpMixer
       │  │  └─IMMPlayableDevice
       │  │     ├─IMMFileMedia
       │  │     │  ├─IMMRecordable
       │  │     │  │  └─IMMConfigurableAudio
       │  │     │  │     ├─IMMDigitalVideo
       │  │     │  │     └─IMMWaveAudio
       │  │     │  └─IMMSequencer
       │  │     └─IMMRemovableMedia
       │  │        ├─IMMAudioCD
       │  │        └─IMMCDXA
       │  └─IMMMasterAudio
       └─IWindow
          └─IControl
             └─ICanvas
                └─IMultiCellCanvas
                   └─IMMPlayerPanel
```

New Color Support

The User Interface Class Library has added new support for the setting, querying, and resetting of colors. In the previous releases of the User Interface Class Library, each class that supported the setting and querying of colors contained a ColorArea enumeration along with an implementation of the Iwindow::setColor and IWindow::color functions. We no longer advise using this enumeration and the two corresponding functions and we have removed them from the interface. They still exist for backward compatibility; however, they might be removed in a future release.

The User Interface Class Library defines a new set of functions for easier color manipulation. For example, to set the background color of an IListBox control you now use the following:

```
listBox->setBackgroundColor(IColor::red);
```

To query the background color of the same listbox:

```
IColor color = listBox->backgroundColor();
```

To set the list box back to its default color:

```
listBox->resetBackgroundColor();
```

These new functions for handling colors mean you no longer have to use different ColorArea enumerators for each control class.

Notes: When you use these functions to implement colors, your controls inherit the colors that you specify for their owners if their owners are also their parent. You can override the inherited color by explicitly setting the color for the specific area of a control.

Some classes use the color area name, such as foregroundColor, on an area of the control that is not related to the name. For example, the system container control does not support a border. However, it uses the border color for specific areas of the control, such as the title separator color. Consult the control documentation in your system reference guide for information on what areas of a control use a specific color area name.

Table 14 (Page 1 of 7). Enumerators to New Color Member Functions

Class	ColorArea enumerators	New Color Member Functions
IButton	foreground	foregroundColor setForegroundColor resetForegroundColor
	background	backgroundColor setBackgroundColor resetBackgroundColor
	disabledForeground	disabledForegroundColor setDisabledForegroundColor resetDisabledForegroundColor
	highlightForeground	hiliteForegroundColor setHiliteForegroundColor resetHiliteForegroundColor
	border	borderColor setBorderColor resetBorderColor
ICanvas	background	backgroundColor setBackgroundColor resetBackgroundColor
IContainerControl	foreground	foregroundColor setForegroundColor resetForegroundColor
	background	backgroundColor setBackgroundColor resetBackgroundColor
	border	borderColor setBorderColor resetBorderColor
	highlightForeground	hiliteForegroundColor setHiliteForegroundColor resetHiliteForegroundColor
	highlightBackground	hiliteBackgroundColor setHiliteBackgroundColor resetHiliteBackgroundColor
IEntryField	foreground	foregroundColor setForegroundColor resetForegroundColor

Table 14 (Page 2 of 7). Enumerators to New Color Member Functions

Class	ColorArea enumerators	New Color Member Functions
	background	backgroundColor setBackgroundColor resetBackgroundColor
	disabledForeground	disabledForegroundColor setDisabledForegroundColor resetDisabledForegroundColor
	highlightForeground	hiliteForegroundColor setHiliteForegroundColor resetHiliteForegroundColor
	border	borderColor setBorderColor resetBorderColor
IFrameWindow	foreground	foregroundColor setForegroundColor resetForegroundColor
	background	backgroundColor setBackgroundColor resetBackgroundColor
	disableBackground	disabledBackgroundColor setDisabledBackgroundColor resetDisabledBackgroundColor
	frameBorder	borderColor setBorderColor resetBorderColor
	shadow	shadowColor setShadowColor resetShadowColor
	activeBorder	activeColor setActiveColor resetActiveColor
	inactiveBorder	inactiveColor setInactiveColor resetInactiveColor
IGroupBox	foreground	foregroundColor setForegroundColor resetForegroundColor

Table 14 (Page 3 of 7). Enumerators to New Color Member Functions

Class	ColorArea enumerators	New Color Member Functions
	border	borderColor setBorderColor resetBorderColor
IListBox	foreground	foregroundColor setForegroundColor resetForegroundColor
	background	backgroundColor setBackgroundColor resetBackgroundColor
	disabledForeground	disabledForegroundColor setDisabledForegroundColor resetDisabledForegroundColor
	highlightForeground	hiliteForegroundColor setHiliteForegroundColor resetHiliteForegroundColor
	border	borderColor setBorderColor resetBorderColor
IMenu	foreground	foregroundColor setForegroundColor resetForegroundColor
	background	backgroundColor setBackgroundColor resetBackgroundColor
	disableForeground	disabledForegroundColor setDisabledForegroundColor resetDisabledForegroundColor
	disableBackground	disabledBackgroundColor setDisabledBackgroundColor resetDisabledBackgroundColor
	highlightForeground	hiliteForegroundColor setHiliteForegroundColor resetHiliteForegroundColor
	highlightBackground	hiliteBackgroundColor setHiliteBackgroundColor resetHiliteBackgroundColor

Table 14 (Page 4 of 7). Enumerators to New Color Member Functions

Class	ColorArea enumerators	New Color Member Functions
	border	borderColor setBorderColor resetBorderColor
IMultiLineEdit	foreground	foregroundColor setForegroundColor resetForegroundColor
	background	backgroundColor setBackgroundColor resetBackgroundColor
	bordercolor	borderColor setBorderColor resetBorderColor
INotebook	pageBackground	pageBackgroundColor setPageBackgroundColor resetPageBackgroundColor
	majorTabForeground	majorTabForegroundColor setMajorTabForegroundColor resetMajorTabForegroundColor
	majorTabBackground	majorTabBackgroundColor setMajorTabBackgroundColor resetMajorTabBackgroundColor
	minorTabForeground	minorTabForegroundColor setMinorTabForegroundColor resetMinorTabForegroundColor
	minorTabBackground	minorTabBackgroundColor setMinorTabBackgroundColor resetMinorTabBackgroundColor
	notebookWindowBackground	backgroundColor setBackgroundColor resetBackgroundColor
	notebookOutline	borderColor setBorderColor resetBorderColor
	statusLineText	foregroundColor setForegroundColor resetForegroundColor

Table 14 (Page 5 of 7). Enumerators to New Color Member Functions

Class	ColorArea enumerators	New Color Member Functions
	selectionCursor	hiliteBackgroundColor setHiliteBackgroundColor resetHiliteBackgroundColor
IOutlineBox	fillRegion	foregroundColor setForegroundColor resetForegroundColor
IProgressIndicator	foreground	foregroundColor setForegroundColor resetForegroundColor
	background	backgroundColor setBackgroundColor resetBackgroundColor
	border	borderColor setBorderColor resetBorderColor
IScrollBar	shaft	foregroundColor setForegroundColor resetForegroundColor
	scrollBox	hiliteForegroundColor setHiliteForegroundColor resetHiliteForegroundColor
	border	borderColor setBorderColor resetBorderColor
ISetCanvas	background	backgroundColor setBackgroundColor resetBackgroundColor
ITextSpinButton INumericSpinButton	foreground	foregroundColor setForegroundColor resetForegroundColor
	disabledForeground	disabledForegroundColor setDisabledForegroundColor resetDisabledForegroundColor
	highlightForeground	hiliteForegroundColor setHiliteForegroundColor resetHiliteForegroundColor

Table 14 (Page 6 of 7). Enumerators to New Color Member Functions

Class	ColorArea enumerators	New Color Member Functions
	border	borderColor setBorderColor resetBorderColor
ISplitCanvas	splitBarEdge	splitBarEdgeColor setSplitBarEdgeColor resetSplitBarEdgeColor
	splitBarMiddle	splitBarMiddleColor setSplitBarMiddleColor resetSplitBarMiddleColor
IStaticText	foreground	foregroundColor setForegroundColor resetForegroundColor
	background	backgroundColor setBackgroundColor resetBackgroundColor
	fill	fillColor setFillColor resetFillColor
ITitle	activeFill	activeColor setActiveColor resetActiveColor
	inactiveFill	inactiveColor setInactiveColor resetInactiveColor
	activeTextForeground	activeTextForegroundColor setActiveTextForegroundColor resetActiveTextForegroundColor
	inactiveTextForeground	inactiveTextForegroundColor setInactiveTextForegroundColor resetInactiveTextForegroundColor
	activeTextBackground	activeTextBackgroundColor setActiveTextBackgroundColor resetActiveTextBackgroundColor
	inactiveTextBackground	inactiveTextBackgroundColor setInactiveTextBackgroundColor resetInactiveTextBackgroundColor

Table 14 (Page 7 of 7). Enumerators to New Color Member Functions

Class	ColorArea enumerators	New Color Member Functions
	border	borderColor setBorderColor resetBorderColor
IViewPort	fillBackground	backgroundColor setBackgroundColor resetBackgroundColor

Tasks and Samples Cross-Reference Table

The following table contains a list of User Interface Class Library tasks and cross-references them to the samples and examples in the *User Interface Class Library User's Guide* that show you how to complete the task.

Tasks	Sample or Example	Class Usage
Create a basic frame window with a line of text centered in the middle of the window	Hello World version 1	IFrameWindow IStaticText
Align a text string within a static text control	Hello World version 2	IStaticText
Display information about an application in an information area below the client window.	Hello World version 2	IFrameExtension IInfoArea
Title your application using a .RC file	Hello World version 3	IFrameWindow
Display application status in a status area.	Hello World version 3	IFrameExtension IFrameWindow IStaticText
Process menu bar items	Hello World version 3	ICommandHandler IMenuBar
Let the user perform functions using accelerator keys	Hello World version 4	IFrameWindow IMenuBar
Request text information from users using a modal dialog	Hello World versions 4, 5, and 6	IFrameWindow
Display multiple controls in a client area using a canvas control	Hello World version 5 Split Canvas Sample	IListBox ISplitCanvas IStaticText
Display information to a user in a list	Hello World version 5 List Box Sample	IListBox
Perform an action when a user selects an item from a list	Hello World version 5	IListBox IListBox::Cursor ISelectHandler

Tasks	Sample or Example	Class Usage
Display and repaint a static text window.	Hello World version 5	IPaintHandler IStaticText
Add help to an application (requires use of IPFX/X on AIX)	Hello World version 5	IHelpHandler IHelpWindow
Perform an action when an user selects an item from a combo box in a dialog window.	Hello World version 6	IComboBox ICommandHandler IFrameWindow
Show multiple components on a status line	Hello World version 6	IFrameExtension ISplitCanvas
Display a digital clock (long and short form)	Hello World version 6	IString ITime
Display the date (long and short form)	Hello World version 6	IDate IString
Let the user dynamically change the font of an application using a font dialog	Hello World version 6	IFontDialog
Customize an application using a notebook control	Hello World version 6	ICommandHandler IFrameWindow INotebook
Customize an application using a pop-up menu	Hello World version 6	IMenuHandler IPopUpMenu
Record a user’s settings of an application	Hello World version 6	IProfile
Customize a control	See the Adding Styles section.	IBitFlag IWindow
Creating a Message Box	See Specifying Message Box Information Message Box Sample	IMessageBox
Save data from a user-edited control to a file	See Viewing and Editing Multiple-Line Edit (MLE) Fields.	IMultiLineEdit
Display a list of choices to a user	See Creating and Using List Controls.	IListBox ISelectHandler
Displaying a discrete set of choices to a user	See Creating and Using List Controls.	INumericSpinButton ITextSpinButton

Tasks	Sample or Example	Class Usage
Arrange child controls in rows or columns	See Creating and Using Canvas Controls. Set Canvas Sample	ISetCanvas
Arrange child controls in a grid of rows and columns	See Understanding Multiple-Cell Canvas Multi-Cell Canvas Sample	IMultiCellCanvas
Provide a scrollable view area	See Understanding View Ports. View Port Sample	IViewPort
Let the user make a file selection using a file dialog control	See Creating and Using File and Font Dialogs.	IFileDialog IFileDialog::Settings
Create pop-up menus	See Creating Menu Bars and Pull-Down Submenus.	IMenuHandler IPopUpMenu
Create and populate a container control	See Creating and Using Containers. Container Sample	ICNRAllocator IContainerControl IContainerObject
Enabling direct manipulation support	See Supporting Direct Manipulation.	IDM* classes
Enable direct manipulation for an entry field or an MLE control.	Direct Manipulation Sample 1	IDmHandler IEntryField IMultiLineEdit
Enabling a bitmap control as a drop target.	Direct Manipulation Sample 2	IBitmapControl IDMHandler IDMItem IDMItemProviderFor IDMTargetDropEvent IDMTargetEvent
Enable direct manipulation for intra-process (source and target containers in the same process) container support.	Direct Manipulation Sample 3	IContainerColumn IContainerControl IContainerObject IDMHandler
Enable direct manipulation for inter-process (source and target containers in the separate processes) container support.	Direct Manipulation Sample 4	IContainerControl IContainerObject IDMCnrItem IDMHandler IDMSourceDiscardEvent IDMSourceOperation IDMTargetDropEvent

Tasks	Sample or Example	Class Usage
Enabling a static text control as a drag source	See Enabling a Control as a Drag Source.	IDMHandler IDMItem IDMItemProvider IDMItemProviderFor IDMSourceOperation IStaticText
Process different events (time changes on a clock)	See Extending Event Handling.	IHandler
Adding mouse handlers	See Handling Mouse Events.	IMouseHandler
Using clipboards in your applications	See Adding Clipboard Support. Container Clipboard Sample	IClipboardHandler IClipboard
Provide toolbars that the user can manipulate and customize	See Adding Tool Bars. Tool Bar Sample	IToolBar IFlyOverHelp
Adding 2-dimensional graphics support to your applications	See Using Graphics in Your Application. 2D Graphics Sample	IG* Classes
Create a remote control interface using multimedia classes.	See Creating and Using Multimedia Controls. Multimedia Sample 1	IMM* classes
Create a stereo interface using multimedia classes.	See Creating and Using Multimedia Controls. Multimedia Sample 2	IMM* classes
Use a multi-line entry field to create a simple editor.	Multi-line Entry Field Sample	ICommandHandler IFileDialog IFontDialog IFrameWindow IHandler IMenuHandler IMultiLineEdit
Use a view port canvas displaying a bitmap as a page in a notebook control	Notebook Sample	IBitmapControl IBitmapHandle INotebook IViewPort

Note: All samples are located in the following directory:

```
\ibmcpp\samples\ioc
```

Using Extended Style Support

The extended style support provides solutions to a number of window style issues when using the User Interface Class Library:

- Overlapping style bit definitions for both OS/2 Presentation Manager
- User Interface Class Library usage of reserved window style bits in the Os/2 Version 2.x operating system (that may be used in a future release of the platform)
- Usage of unused window style bits to represent other style bits (for example, IStaticText).
- Extended style support implemented directly within the class (for example, the canvas classes)
- No simple method to extend existing control set and add extended style support, such as building a composite custom control.
- Portability concerns

The objectives of the new extended style support are as follows:

- No breakage of the existing styles support
- Easy integration with the existing styles support
- Reserved published ranges for the User Interface Class Library, as well as customer extended styles bits
- Assist porting efforts

The key function `convertToGUIStyle( const IBitFlag& style, Boolean extended )` allows the individual classes to accept a style object, and resolve the base and extended styles that it contains into a window style. You pass this window style to the underlying platform during window creation.

The key parameter, *const IBitFlag& style*, exposes the IBitFlag class, because you cannot truly inherit a style class. For example, ICanvas derived classes use IControl styles due to friendship, not inheritance.

Currently, we reserve the extended style bits as follows:

> Bits 0-23 are reserved for the User Interface Class Library
> Bits 24-31 are open for your usage

The User Interface Class Library requires 24-Bits due to the proliferation of styles in ISetCanvas and ISetCanvas derived classes such as the tool bar classes.

Base styles are those window styles that are predefined for each control on the supported platform. An example of a base style would be the OS/2 Presentation Manager window style, WS_VISIBLE. WS_VISIBLE is exposed in the User Interface Class Library as IWindow::visible.

The following is an example of initializing a base style:

```
const IWindow::Style IWindow::visible = WS_VISIBLE;
```

Extended styles define styles that you do not see predefined for the supported platform. The User Interface Class Library has enhanced the extended styles definitions to handle the following scenarios:

- Definition of an extended style where the base style is defined to be 0. An example is the OS/2 Presentation Manager button style, BS_PUSHBUTTON. The User Interface Class Library defines a unique nonzero bit value that allows you to safely use the bitwise operator, &, to test for the existence of this extended style rather than using a masking and comparison technique that you would normally perform against the base style.
- Definition of an extended style where the base style bit value overlaps with another style. An example is the OS/2 Presentation Manager entryfield style, ES_MIXED. The User Interface Class Library defines a unique bit value that allows you to safely use the bitwise operator, &, to test for the existance of this extended style rather than using a comparison technique that you would normally perform against the base style. Note also that the overlapping bits create bitwise testing problems for the entryfield styles that they overlap: ES_SBCS and ES_DBCS.

The following is an example of initializing an extended style:

```
const ISetCanvas::Style ISetCanvas::packTight ( 0, ICNV_PACKTIGHT );
```

The following sections discuss the classes that we modified to add the extended styles support.

IBitFlag

The IBitFlag class was modified to accommodate the extended styles support. The following function is added:

asExtendedULong
Converts the upper 32-Bits of the object to an unsigned long value.

For example:

```
unsigned long
  asExtendedUnsignedLong ( ) const;
```

Additionally, the protected IBitFlag constructor was modified to add a parameter to represent the extended style, *extendedValue*. Note how a default value is being assigned to allow existing style implementations to work:

```
IBitFlag       ( unsigned long value,
                 unsigned long extendedValue = 0 );
```

The protected function, setValue, was modified to support extended styles as well. Again, note the default value assigned to the parameter, *extendedValue*, that allow existing style implementations to work:

```
IBitFlag
 &setValue       ( unsigned long value ,
                   unsigned long extendedValue = 0 );
```

IWindow

To parallel the functions style and setStyle, two new protected functions are added to support extended styles:

extendedStyle
Returns an unsigned long representing the window's extended style.

setExtendedStyle
Sets the extended window style.

```
virtual IWindow
 &setExtendedStyle ( unsigned long extendedStyle );
virtual unsigned long
  extendedStyle    ( ) const;
```

To use extended styles, use the following steps in your constructors to properly save the extended styles:

1. Query the existing existing extended styles using IWindow::extendedStyle. Use IMenuItem::extendedStyle for menu items.

2. Use the bitwise OR operator (|) to include your extended styles with the existing extended styles.

3. Set the new extended styles using IWindow::setExtendedStyle. Use IMenuItem::setExtendedStyle for menu items.

For example:

```
FooClass :: FooClass( unsigned long     ulId,
                      IWindow*          pParent,
                      IWindow*          pOwner,
                      const IRectangle& rectInit,
                      const Style&      style )
   : IControl( )
```

```
  {
    // Save the extended style to make sure a copy of it stored
    setExtendedStyle( extendedStyle() | style.asExtendedUnsignedLong() );
⋮
  }
```

To support both the base and extended styles, another virtual function was added that has the ability to convert the base or extended styles into a style that can be understood by the underlying platform:

convertToGUIStyle

Use this function to convert style bits into the style value that can be processed by the GUI. The default action is to return the base GUI style for the platform. Extended styles which are defined by the User Interface Class Library, can be returned by setting the *extended* parameter to true.

For example:

```
virtual unsigned long
  convertToGUIStyle  ( const IBitFlag& style,
                       Boolean         extended = false ) const;
```

The first parameter is a reference to a style object that you pass on various User Interface Class Library constructors. Use the second parameter to return the style information from the extended portion of the style object. The second parameter can simplify processing of the extended style in your classes. An User Interface Class Library example is IFrameWindow, as we use it to return information on the frame control flags which are passed as an extended style.

The intent of convertToGUIStyle is to allow you to override it throughout the IWindow class hierarchy to allow your classes to parse style information that is pertinent to the creation of your window. Note that we implement the convertToGUIStyle function in this fashion within the User Interface Class Library, and have listed our classes below.

For example:

```
FooClass :: FooClass( unsigned long     ulId,
                      IWindow*          pParent,
                      IWindow*          pOwner,
                      const IRectangle& rectInit,
                      const Style&      style )
   : IControl( )
{
  // Save the extended style to make sure a copy of it stored
  setExtendedStyle( extendedStyle() | style.asExtendedUnsignedLong() );
  //
  IWindowHandle whFooControl =
      create( ulId,
              0,
              convertToGUIStyle( style ),
              "FooWindow"
              pParent->handle(),
              (pOwner == 0) ? IWindowHandle(0) : pOwner->handle(),
              rectInit,
              0,
              0 );
⋮
 }
⋮
unsigned long FooClass :: convertToGUIStyle( const IBitFlag& guiStyle,
                                             Boolean bExtOnly ) const
{
  // Obtain the style from the class (IControl) that we inherit from
  unsigned long ulStyle = Inherited::convertToGUIStyle( guiStyle, bExtOnly );

  if (bExtOnly)
  {
    // Use mask to only return extended styles in the user defined range
    ulStyle |= extendedStyle() & IS_EXTMASK;
  }
  else
  {
    // Mask out FOO_
    ulStyle |= guiStyle.asUnsignedLong() & FOO_MASK;
  }

  return( ulStyle );
}
⋮
```

IMenuItem

The class IMenuItem is a special case. We implement the extended styles support in IMenuItem that is present in IWindow. IMenuItem is not derived from IWindow. Therefore, the extended styles implementation we describe for IWindow applies to IMenuItem as well.

Classes that Implement or Override the convertToGUIStyle Function

IFileDialog
IFontDialog
IFrameWindow
IMenu
IMenuBar

IMenuItem
IWindow

I3StateCheckBox
IBitmapControl
ICanvas
ICheckBox
IComboBox
IDrawingCanvas
IEntryField
IGraphicPushButton
IGroupBox
IIconControl
IListBox
IMultiCellCanvas
IMultiLineEdit
INotebook
IOutlineBox
IPushButton
IRadioButton
IScrollBar
ISetCanvas
IProgressIndicator
INumericSpinButton
ITextSpinButton
ISplitCanvas
IStaticText
IViewPort

IContainerControl

IAnimatedButton
ICustomButton
IToolBar
IToolBarButton
IToolBarContainer

ICircularSlider

Obsolete and Ignored Members Cross-Reference Tables

This appendix contains the following cross-reference tables:

"Obsolete Classes and Members"
Lists obsolete classes and members, and their replacements, if any.

"Ignored Classes and Members" on page 686
Lists classes and members that the User Interface Class Library for AIX does not support and subsequently ignores.

Obsolete Classes and Members

Lists obsolete classes and members, and their replacements, if any.

The User Interface Class Library has added new support for the setting, querying, and resetting of colors. See Appendix B, "New Color Support" on page 665 for information on obsolete and replacement members for colors.

Class	Member	Replacement
IAccelerator	unset	reset
IContainerColumn	isHeadingReadOnly isReadOnly	isHeadingWriteable isWriteable
IContainerControl	detailObjectRectangle (both versions) isReadOnly	detailsObjectRectangle isWriteable
IContainerObject	iconOffset iconTextOffset isReadOnly	None None isWriteable
ICurrentThread	initializePM isPMInitialized terminatePM	initializeGUI isGUIInitialized terminateGUI
IDMTargetOperation	setContainerNoRefresh setContainerRefresh	IDMOperation::setContainerRefreshOff IDMOperation::setContainerRefreshOn
IEntryField	isReadOnly	isWriteable
IFont	enum Direction bottomTop enum Direction leftRight enum Direction rightLeft enum Direction topBottom	enum Direction bottomToTop enum Direction leftToRight enum Direction rightToLeft enum Direction topToBottom

Class	Member	Replacement
IFrameWindow	flagsFrom	convertToGUIStyle
	styleFrom	convertToGUIStyle
IHelpHandler	hypertextSelect	hyperlinkSelect
IHelpHypertextEvent		IHelpHyperlinkEvent
IHelpWindow	associateWindow	setAssociatedWindow
	helpForWindow	None
IListBox	setHeight	setItemHeight
IListBoxDrawItemHandler	draw	drawItem
	drewSelected	setSelectionStateDrawn
	highlight	selectItem
	setHeight	setItemSize
	unhighlight	deselectItem
IMenu	addAt	add
	addNextAt	addAsNext
	isItemDisabled	isItemEnabled
	isVerticalFlip	None
	setVerticalFlip	None
	verticalFlip	None
IMouseClickHandler		IMouseHandler
IMultiLineEdit	cursor	cursorPosition
	disableRefresh	disableUpdate
	enableRefresh	enableUpdate
	isReadOnly	isWriteable
	removeChangedFlag	resetChangedFlag
	setCursorAt	setCursorPosition
	setCursorAtLine	setCursorLinePosition
IPushButton	disableBorder	removeBorder
	enableBorder	addBorder
	isBorder	hasBorder
ISpinButton		IBaseSpinButton INumericSpinButton ITextSpinButton
ISubmenu	addAt	add
	addNextAt	addAsNext
IThread	autoInitPM	autoInitGUI
	defaultAutoInitPM	defaultAutoInitGUI
	setAutoInitPM	setAutoInitGUI
	setDefaultAutoInitPM	setDefaultAutoInitGUI

Class	Member	Replacement
ITitle	setViewNum viewNum	setViewNumber viewNumber
IWindow	handleWithId isDisabled windowWithId	handleWithParent isEnabled windowWithOwner

Ignored Classes and Members

Lists classes and members that the User Interface Class Library for AIX does not support and subsequently ignores.

A single asterisk (*) means this member is overloaded and you must look it up to determine which overload the AIX release ignores.

A double asterisk (**) means this class is not ported to the AIX platform.

Table 15 (Page 1 of 14). Ignored members in Motif

Class	Member
I3StateCheckBox **	
I3StateCheckBox::Style **	
IAccelerator	handle isSet remove reset set *
IAccelTblHandle	
IAnchorBlockHandle	
IBaseSpinButton	alignment setAlignment
IButton	allowsMouseClickFocus disableMouseClickFocus enableMouseClickFocus highlight isHighlighted unhighlight
ICnrBeginEditEvent **	
ICnrDrawBackgroundEvent **	
ICnrDrawHandler **	
ICnrDrawItemEvent **	
ICnrEditEvent **	
ICnrEditHandler **	
ICnrEndEditEvent **	
ICnrEnterEvent	

Table 15 (Page 2 of 14). Ignored members in Motif

Class	Member
ICnrHandler	cursoredChanged deltaReached inuseChanged windowScrolled
ICnrMenuHandler	addSourceEmphasis removeSourceEmphasis
ICnrQueryDeltaEvent **	atBottomDelta atFirstObject atLastObject atTopDelta ICnrQueryDeltaEvent ˜ICnrQueryDeltaEvent
ICnrReallocStringEvent **	
ICnrScrollEvent	amount ICnrScrollEvent isHorizontal isLeftDetails isRightDetails isVertical ˜ICnrScrollEvent
IContainerColumn	dataAttributes disableDataUpdate disableHeadingUpdate displayWidth enableDataUpdate enableHeadingUpdate headingIcon hideSeparators horizontalDataAlignment horizontalHeadingAlignment isHeadingIconHandle isHeadingReadOnly isHeadingString isHeadingWriteable isReadOnly isWriteable

Table 15 (Page 3 of 14). Ignored members in Motif

Class	Member
IContainerColumn (cont.)	justifyData justifyHeading setDataAttributes setDisplayWidth setHeadingIcon * setTitleAttributes showSeparators titleAttributes verticalDataAlignment verticalHeadingAlignment
IContainerControl	areDetailsViewTitlesVisible closeEdit collapse collapseTree columnUnderPoint convertToWorkspace currentEditColumn currentEditMLE currentEditObject cursoredObject detailObjectRectangle * detailsObjectRectangle * detailsTitleRectangle detailsViewPortOnWindow detailsViewPortOnWorkspace detailsViewSplit disableCaching disableDataUpdate disableDrawBackground disableDrawItem disableDrop disableTitleUpdate editColumnTitle

Table 15 (Page 4 of 14). Ignored members in Motif

Class	Member
IContainerControl (cont.)	editContainerTitle editObject enableCaching enableDataUpdate enableDrawBackground enableDrawItem enableDrop enableTitleUpdate expand expandTree hasMixedTargetEmphasis hasNormalTargetEmphasis hasOrderedTargetEmphasis hideSourceEmphasis * hideSplitBar hideTitle hideTitleSeparator iconRectangle iconSize isCachingEnabled isCollapsed isColumnRight isCursored isDrawBackgroundEnabled isDrawItemEnabled isDropOnAble isExpanded isInUse isReadOnly isRefreshOn isSource isTarget isTitleSeparatorVisible isTitleVisible isTitleWriteable isVisible * isWriteable lineSpacing moveIconTo objectAt *

Table 15 (Page 5 of 14). Ignored members in Motif

Class	Member
IContainerControl (cont.)	objectUnderPoint refresh * refreshAllContainers removeInUse scroll scrollDetailsHorizontally scrollHorizontally scrollToObject * scrollVertically setCursor setDetailsViewSplit setEditColumn setEditMLE setEditObject setIconSize setInUse setLineSpacing setMixedTargetEmphasis setNormalTargetEmphasis setOrderedTargetEmphasis setRefreshOff setRefreshOn setTitle * setTitleAlignment setTreeExpandIconSize showSourceEmphasis * showSplitBar showTitle showTitleSeparator splitBarOffset textRectangle title titleRectangle viewPortOnWindow viewPortOnWorkspace

Table 15 (Page 6 of 14). Ignored members in Motif

Class	Member
IContainerObject	disableDataUpdate disableDrop enableDataUpdate enableDrop handleCursoredChange handleInuseChange iconOffset iconTextOffset isDropOnAble isInUse isReadOnly isRefreshOn isWriteable refresh removeInUse setInUse setRefreshOff setRefreshOn
IControl	disableGroup disableTabStop enableGroup enableTabStop isGroup isTabStop
ICurrentThread	remainingStack suspend waitFor waitForAllThreads waitForAnyThread
IDDEAcknowledgeEvent **	
IDDEAcknowledgeExecuteEvent **	
IDDEAcknowledgePokeEvent **	
IDDEActiveServer **	
IDDEActiveServerSet **	
IDDEBeginEvent **	
IDDEClientAcknowledgeEvent **	
IDDEClientConversation **	
IDDEClientEndEvent **	

Table 15 (Page 7 of 14). Ignored members in Motif

Class	Member
IDDEClientHotLinkEvent **	
IDDEClientHotLinkSet **	
IDDEDataEvent **	
IDDEEndEvent **	
IDDEEvent **	
IDDEExecuteEvent **	
IDDEPokeEvent **	
IDDERequestDataEvent **	
IDDEServerAcknowledgeEvent **	
IDDEServerHotLinkEvent **	
IDDESetAcknowledgeInfoEvent **	
IDDETopicServer **	
IDeviceColor	
IDMCnrItem **	
IDMEFItem **	
IDMEvent **	
IDMHandler **	
IDMImage **	
IDMItem **	
IDMItemProvider **	
IDMItemProviderFor **	
IDMMLEItem **	
IDMOperation **	
IDMRenderer **	
IDMSourceBeginEvent **	
IDMSourceDiscardEvent **	
IDMSourceEndEvent **	
IDMSourceHandler **	
IDMSourceOperation **	

Table 15 (Page 8 of 14). Ignored members in Motif

Class	Member
IDMSourcePrepareEvent **	
IDMSourcePrintEvent **	
IDMSourceRenderer **	
IDMSourceRenderEvent **	
IDMTargetDropEvent **	
IDMTargetEndEvent **	
IDMTargetEnterEvent **	
IDMTargetHandler **	
IDMTargetHelpEvent **	
IDMTargetLeaveEvent **	
IDMTargetOperation **	
IDMTargetRenderer **	
IDrawItemEvent **	
IDynamicLinkLibrary	
IEntryField	setAlignment setCharType
IEnumHandle	
IFileDialog::Settings	addDrive addFileType setInitialDrive setInitialFileType
IFont	isOutline isStrikeout isUnderscore setCharHeight setCharSize setCharWidth setDirection setFontAngle setFontShear setOutline setStrikeout setUnderscore
IFont::FaceNameCursor	

Table 15 (Page 9 of 14). Ignored members in Motif

Class	Member
IFont::PointSizeCursor	
IFontDialog	fontWeight fontWidth
IFontDialog::Settings	setDisplayPS setPrinterPS
IFrameFormatEvent	clientRect IFrameFormatEvent setClientRect swpArray ˜IFrameFormatEvent
IFrameHandler	deactivated draw format positionExtensions
IFrameWindow	addToWindowList beginFlashing borderHeight borderSize borderWidth endFlashing isFlashing isMaximized maximize maximizeRect minimizeRect nextShellRect removeFromWindowList restore restoreRect setBorderHeight setBorderSize * setBorderWidth setRestoreRect shareParentDBCSStatus usesDialogBackground
IGraphicPushButton	disableSizeToGraphic enableSizeToGraphic marginSize setMarginSize

Table 15 (Page 10 of 14). Ignored members in Motif

Class	Member
IGroupBox	calcMinimumSize defaultStyle foregroundColor moveSizeTo position rect setDefaultStyle setText * size
IGroupBox::Style	
IGUIColor	
IHelpHandler	openLibrary showContents showCoverPage showHistory showIndex showPage showSearchList swapPage
IInfoArea	setResourceLibrary *
IListBox	disableNoAdjustPosition enableNoAdjustPosition isNoAdjustPosition
IListBoxDrawItemEvent **	
IListBoxDrawItemHandler **	
IListBoxDrawItemHandler::DrawFlag **	
IMenu	removeConditionalCascade setConditionalCascade
IMenuDrawItemEvent **	
IMenuDrawItemHandler **	
IMenuDrawItemHandler::DrawFlag **	

Table 15 (Page 11 of 14). Ignored members in Motif

Class	Member
IMenuItem	isDrawItem isFramed isHighlighted isNoDismiss setDrawItem setFramed setHighlighted setNoDismiss
IMessageQueueHandle	
IModuleHandle	
IMultiCellCanvas	disableDragLines disableGridLines enableDragLines enableGridLines hasDragLines hasGridLines
IMultiLineEdit	isUndoable setTab undo
INotebook	hiliteBackgroundColor resetBackgroundMajorColor resetBackgroundMinorColor resetBackgroundPageColor resetForegroundMajorColor resetForegroundMinorColor setMajorTabSize setMinorTabSize setPageButtonSize setTabShape tabShape
INotebookDrawItemEvent **	
INotifier	
IPageHandler	help remove resize
IPageHelpEvent	helpWindow IPageHelpEvent * notebook pageHandle ~IPageHelpEvent

Table 15 (Page 12 of 14). Ignored members in Motif

Class	**Member**
IPageRemoveEvent	IPageRemoveEvent * notebook pageWindow tabBitmap ˜IPageRemoveEvent
IProcedureAddress	
IProfile	numberOfApplications
IProgressIndicator	disableDrawItem enableDrawItem isDrawItemEnabled setShaftPosition
IRadioButton	disableAutoSelect disableCursorSelect enableAutoSelect enableCursorSelect isAutoSelect isCursorSelect
IRegionHandle	
ISlider	addDetent detentPosition removeDetent setArmSize
ISliderDrawHandler **	
ISpinButton	alignment setAlignment
ISplitCanvas	resetSplitBarEdgeColor resetSplitBarMiddleColor setSplitBarEdgeColor splitBarEdgeColor
ISWP	
ISWPArray	
ISystemBitmapHandle **	
ISystemMenu	ISystemMenu *

Table 15 (Page 13 of 14). Ignored members in Motif

Class	Member
IThread	adjustPriority defaultQueueSize defaultStackSize priorityClass priorityLevel queueSize resume setDefaultQueueSize setDefaultStackSize setPriority setQueueSize setStackSize stackSize suspend
ITitle	activeColor activeTextBackgroundColor activeTextForegroundColor borderColor inactiveColor inactiveTextBackgroundColor inactiveTextForegroundColor resetActiveTextBackgroundColor resetActiveTextForegroundColor resetInactiveTextBackgroundColor resetInactiveTextForegroundColor setActiveTextBackgroundColor setActiveTextForegroundColor setInactiveTextBackgroundColor setInactiveTextForegroundColor

Table 15 (Page 14 of 14). Ignored members in Motif

Class	Member
IWindow	activeColor disabledBackgroundColor disabledForegroundColor dispatch hiliteBackgroundColor hiliteForegroundColor inactiveColor resetActiveColor resetBackgroundColor resetBorderColor resetDisabledBackgroundColor resetDisabledForegroundColor resetForegroundColor resetHiliteBackgroundColor resetHiliteForegroundColor resetInactiveColor resetShadowColor setActiveColor setDisabledBackgroundColor setDisabledForegroundColor setHiliteBackgroundColor setHiliteForegroundColor setId setInactiveColor setParent setShadowColor setStyle shadowColor style

Glossary

This glossary defines terms and abbreviations that are used in this book. If you do not find the term you are looking for, refer to the *IBM Dictionary of Computing*, New York:McGraw-Hill, 1994.

This glossary includes terms and definitions from the *American National Standard Dictionary for Information Systems*, ANSI X3.172-1990, copyright 1990 by the American National Standards Institute (ANSI). Copies may be purchased from the American National Standards Institute, 1430 Broadway, New York, New York 10018.

A

abstract class. (1) A class with at least one pure virtual function that is used as a base class for other classes. The abstract class represents a concept; classes derived from it represent implementations of the concept. You cannot construct an object of an abstract class. See also base class. (2) A class that allows polymorphism.

abstract data type. A mathematical model that includes a structure for storing data and operations that can be performed on that data. Common abstract data types include sets, trees, and heaps.

abstraction (data). See data abstraction.

access. An attribute that determines whether or not a class member is accessible in an expression or declaration. It can be public, protected, or private.

access declaration. A declaration used to adjust access to members of a base class.

access function. A function that returns information about the elements of an object so that you can analyze various elements of a string.

access resolution. The process by which the accessibility of a particular class member is determined.

access specifier. One of the C++ keywords public, private, or protected.

ambiguous derivation. A derivation where the class is derived from two or more base classes that have members with the same name.

amplifier. A device that increases the strength of input signals. Also referred to as an amp.

amplifier-mixer. A combination amplifier and mixer that is used to control the characters of an audio signal from one or more audio sources. Also referred to as an amp-mixer.

animate. Make or design in such a way as to create apparently spontaneous, lifelike movement.

animation rate. The number of thousandths of a second that pass before the next bitmap is displayed for a button while it is animated.

anonymous union. A union that is declared within a structure or class and that does not have a name.

area. In computer graphics, a filled shape, such as a solid rectangle.

array. An aggregate that consists of data objects, with identical attributes, each of which may be uniquely referenced by subscripting.

array implementation. (In Collection Class Library) Implementation of an abstract data type using an array. Also called a tabular implementation.

ASCII (American National Standard Code for Information Interchange). The standard code, using a coded character set consisting of 7-bit coded characters (8 bits including parity check), that is used for information interchange among data processing systems, data communication systems, and associated equipment. The ASCII set consists of control characters and graphic characters.

Note: IBM has defined an extension to ASCII code (characters 128-255).

audio. Pertaining to the portion of recorded information that can be heard.

audio attributes. The standard audio attributes are: mute, volume, balance, treble, and bass.

audio formats. The way the audio information is stored and interpreted.

audio track. (1) The audio (sound) portion of the program. (2) The physical location where the audio is places beside the image. (A system with two sound tracks can have either stereo sound or two independent sound tracks.) Synonymous with sound track.

automatic storage. Storage that is allocated on entry to a routine or block and is freed on the subsequent return. Sometimes referred to as *stack storage* or *dynamic storage*.

automatic storage management. The process that automatically allocates and deallocates objects in order to use memory efficiently.

auxiliary classes. Classes that support other classes. Auxilliary classes in the Collection Class Library include classes for cursors, pointers and iterators.

AVL tree. A balanced binary search tree that does not allow the height of two siblings to differ by more than one.

B

B*-tree (B star tree). A tree in which only the leaves contain whole elements. All other nodes contain keys.

background color. The color in which the background of a graphic primitive is drawn.

balance. (1) For audio, refers to the relative strength of the left and right channels. A balance level of 0 is left channel only. A balance level of 100 is right channel only (2) A state of equilibrium, usually between treble and bass.

base class. A class from which other classes are derived. A base class may itself be derived from another base class. See also abstract class.

based on. A relationship between two classes in which one class is implemented through the other. A new class is “based on” an existing class when the existing class is used to implement it.

bass. The lower half of the whole vocal or instrumental tonal range.

bit field. A member of a structure or union that contains a specified number of bits.

bit mask. A pattern of characters used to control the retention or elimination of portions of another patterns of characters.

bits-per-sample. The number of bits of audio data that is to represent each sample of each channel (right or left). This is the resolution of the audio data. CD quality needs to be 16 bits-per-sample.

boundary alignment. The position in main storage of a fixed-length field (such as byte or doubleword) on an integral boundary for that unit of information.

For the Class Library example, a word boundary is a storage address evenly divisible by two.

bounded collection. A collection that has an upper limit on the number of elements it can contain.

brightness. The level of luminosity of the video signal. A brightness level of 0 produces a maximally white signal. A brightness level of 100 produces a maximally black signal.

built-in. A function that the compiler automatically puts inline instead of generating a call to the function.

C

camcorder. A compact, hand-held video camera with integrated videotape recorder.

canvas. Canvases are windows with a layout algorithm that manage child windows. The canvas classes are a set of window classes which allow you to implement dialog-like windows (that is, a window with several child controls). These windows are used for showing views of objects as both pages in a notebook and as windows that gather information to run an action. The different canvases can manage the size and position of child windows, provide moveable split bars between windows, and support the ability to scroll a window.

The canvases include the base class, ICanvas, and its four derived classes: IMultiCellCanvas, ISetCanvas, ISplitCanvas, and IViewport.

cast. A notation used to express the conversion of one type to another.

catch block. A block associated with a try block that receives control when a C++ exception matching its argument is thrown.

CD. Compact disc

CD-ROM. Compact disc-read-only memory

CD-XA. Compact disc-extended architecture

channel mapping. The translation of a MIDI channel number for a sending device to an appropriate channel for a receiving device.

character array. An array of type char.

child. A node that is subordinate to another node in a tree structure. Only the root node of a tree is not a child.

child class. See derived class.

child window. A window derived from another window and drawn relative to it.

circular slider control. A 360-degree knob-like control that simulates the buttons on a TV, a stereo, or video components. By rotating the slider arm, the user can set, display, or modify a value, such as the balance, bass, volume, or treble.

class. A user-defined type. Classes can be defined hierarchically, allowing one class to be an expansion of another, and classes can restrict access to their members.

class hierarchy. A tree-like structure showing relationships among classes. It places one abstract class at the top (a base class) and one or more layers of derived classes below it.

class library. A collection of classes.

class template. A blueprint describing how a set of related classes can be constructed.

client area window. An intermediate window between an IFrameWindow and its controls and other child windows.

client program. A program that uses a class. The program is said to be a client of the class.

collection. (1) In a general sense, an implementation of an abstract data type for storing elements. (2) An abstract class without any ordering, element properties, or key properties. All abstract Collection Classes are derived from Collection.

Collection Classes. A set of classes that implement abstract data types for storing elements.

color palette. A set of all the colors that can be used in a displayed image.

compact disc (CD). (1) A disc, usually 4.75 inches in diameter, from which data is read optically by means of a laser. (2) A disc with information stored in the form of pits along a spiral track. The information is decoded by a compact-disc player and interpreted as digital audio data, which most computers can process.

compact disc-extended architecture (CD-EX). A storage format that accommodates interleaved storage of audio, video, and standard file system data.

compact disc-read-only memory (CD-ROM). (1) An optical storage medium (2) High-capacity, read-only memory in the form of an optically read compact disc.

Complex Mathematics library. A C++ class library that provides the facilities to manipulate complex numbers and perform standard mathematical operations on them.

composite. The combination of two or more film, video, or electronic images into a single frame or display.

computer-controlled device. An external video source device with frame-stepping capability, usually a videodisc player, whose output can be controlled by the multimedia subsystem.

concrete class. A class that implements an abstract data type but does not allow polymorphism.

const. (1) An attribute of a data object that declares that the object cannot be changed. (2) An attribute of a function that declares that the function will not modify data members of its class.

constructor. A special class member function that has the same name as the class and is used to construct and possibly initialize objects of its class type. A return type is not specified.

containment function. A function that determines whether a collection contains a given element.

copy constructor. A constructor used to make a copy of an object from another object of the same type.

critical section. Code that must be executed by one thread while all other threads in the process are suspended.

cursor. A reference to an element at a specific position in a data structure.

cursor iteration. The process of repeatedly moving the cursor to the next element in a collection until some condition is satisfied.

cursored emphasis. When the selection cursor is on a choice, that choice has cursored emphasis.

C/2. A version of the C language designed for the OS/2 environment.

D

daemon. A program that runs unattended to perform a service for other programs.

data abstraction. A data type with a private representation and a public set of operations. The C++ language uses the concept of classes to implement data abstraction.

DBCS (Double-Byte Character Set). See double-byte character set.

deck. A line of child windows in a set canvas that is direction-independent. A horizontal deck is equivalent to a row and a vertical deck is equivalent to a column.

declaration. Introduces a name to a program and specifies how the name is to be interpreted.

declare. To specify the interpetation that C++ gives to each identifier.

default argument. An argument that is declared with a default value in a function prototype or declaration. If a call to the function omits this argument, the default value is used. Arguments with default values must be the trailing arguments in a function prototype argument list.

default class. A class with preprogrammed definitions that can be used for simple implementations.

default constructor. A constructor that takes no arguments, or a constructor for which all the arguments have default values.

default implementation. One of several possible implementation variants offered as the default for a specific abstract data type.

default operation class. A class with preprogrammed definitions for all required element and key operations for a particular implementation.

degree. The number of children of a node.

delete. (1) A C++ keyword that identifies a free-storage deallocation operator. (2) A C++ operator used to destroy objects created by operator new.

deque. A queue that can have elements added and removed at both ends. A double-ended queue.

dequeue. An operation that removes the first element of a queue.

derivation. (1) The creation of a new or derived class from an existing base class. (2) The relationship between a class and the classes above or below it in a class hierarchy.

derived class. A class that inherits from a base class. You can add new data members and member functions to the derived class. You can manipulate a derived class object as if it were a base class object. The derived class can override virtual functions of the base class.

Synonym for child class and subclass.

destructor. A special member function that has the same name as its class, preceded by a tilde (˜), and that “cleans up” after an object of that class, for example, by freeing storage that was allocated when the object was created. A destructor has no arguments, and no return type is specified.

difference. Given two sets A and B, the difference (A-B) is the set of all elements contained in A but not in B.

digital audio. Audio data that has been converted to digital form.

digital video. Material that can be seen and that has been converted to digital form.

digital video device. A full-motion video device that can record or play files (or both) containing digitally stored video.

diluted array. An array in which elements are deleted by being flagged as deleted, rather than by actually removing them from the array and shifting later elements to the left.

diluted sequence. A sequence implemented using a diluted array.

direct manipulation. A user interface technique whereby the user initiates application functions by manipulating the objects, represented by icons, on the Presentation Manager (PM) or Workplace Shell desktop. The user typically initiates an action by:

1. Selecting an icon
2. Pressing and holding down a mouse button while “dragging” the icon over another object’s icon on the desktop
3. Releasing the mouse button to “drop” the icon over the target object.

Thus, this technique is also known as “drag and drop” manipulation.

double-byte character set (DBCS). A set of characters in which each character is represented by 2 bytes. Languages such as Japanese, Chinese, and Korean, which contain more symbols than can be represented by 256 code points, require double-byte character sets.

Because each character requires 2 bytes, you need hardware and supporting software that are DBCS-enabled to enter, display, and print DBCS characters.

doubleword. A contiguous sequence of bits or characters that comprises two computer words and can be addressed as a unit. For the C Set++ for AIX compiler, a doubleword is 32 bits (4 bytes).

drag after. A target enter event that occurs in a container where its orderedTargetEmphasis or mixedTargetEmphasis attribute is set and the current view is name, text, or details.

drag item. A "proxy" for the object being manipulated.

drag over. A target enter event that occurs in a container where its orderedTargetEmphasis attribute is not set and the current view is icon or tree view.

drop offset. The location where the next container object that is dropped will be positioned (if the target operation's drop style is not IDM::dropPosition). The position is based upon the last object that was dropped as an offset of that object relative to the drop style.

E

EBCDIC (extended binary-coded decimal interchange code). A coded character set of 256 8-bit characters.

element. The component of an array, subrange, enumeration, or set.

element equality. A relation that determines whether two elements are equal.

element function. A function, called by a member function, that accesses the elements of a class.

encapsulation. The hiding of the internal representation of objects and implementation details from the client program.

enqueue. An operation that adds an element as the last element to a queue.

enumeration constant. An identifier that is defined in an enumeration and that has an associated constant integer value. You can use an enumeration constant anywhere an integer constant is allowed.

enumeration data type. A type that represents integers and a set of enumeration constants. Each enumeration constant has an associated integer value.

equality collection. (1) An abstract class with the property of element equality. (2) In general, any collection that has element equality.

equality key collection. An abstract class with the properties of element equality and key equality.

equality key sorted collection. An abstract class with the properties of element equality, key equality, and sorted elements.

equality sequence. A sequentially ordered flat collection with element equality.

equality sorted collection. An abstract class with the properties of element equality and sorted elements.

exception. (1) A user or system error detected by the system and passed to an OS/2 or user exception handler. (2) For C++, any user, logic, or system error detected by a function that does not itself deal with the error but passes the error on to a handling routine (also called "throwing the exception").

exception handler. (1) A function that is invoked when an exception is detected, and that either corrects the problem and returns execution to the program, or terminates the program. (2) In C++, a catch block that catches a C++ exception when it is thrown from a function in a try block.

exception handling. A type of error handling that allows control and information to be passed to an exception handler when an exception occurs. Under the OS/2 operating system, exceptions are generated by the system and handled by user code. In C++, try, catch, and throw expressions are the constructs used to implement C++ exception handling.

external data definition. A definition appearing outside a function. The defined object is accessible to all functions that follow the definition and are located within the same source file as the definition.

eyecatcher. A recognizable sequence of bytes that determines which parameters were passed in which registers. This sequence is used for functions that have not been prototyped or have a variable number of parameters.

F

file descriptor. A small positive integer that the system uses instead of the file name to identify an open file.

file scope. A name declared outside all blocks and classes has file scope and can be used after the point of declaration in a source file.

filter. A command whose operation consists of reading data from standard input or a list of input files and writing data to standard output. Typically, its function is to perform some transformation on the data stream.

first element. The element visited first in an iteration over a collection. Each collection has its own definition for first element. For example, the first element of a sorted set is the element with the smallest value.

flat collection. A collection that has no hierarchical structure.

font. A particular size and style of typeface that contains definitions of character sets, marker sets, and pattern sets.

frame. (1) A complete television picture that is composed of two scanned fields, one of the even lines and one of the odd lines. In the NTSC system, a frame has 525 horizontal lines and is scanned in 1/30th of a second. (2) A border around a window.

frame extension. A control you can add if it is not available in the basic Presentation Manager frame windows.

frame number. (1) The number used to identify a frame. (2) The location of a frame on a videodisc or in a video file. On videodisc, frames are numbered sequentially from 1 to 54,000 on each side and can be accessed individually; on videotape, the numbers are assigned by way of the SMPTE time code.

frame rate. The speed at which the frames are scanned. For a videodisc player, the speed at which frames are scanned is 30 frames per second for NTSC video. For most videotape devices, the speed is 24 frames per second.

friend class. A class in which all the member functions are granted access to the private and protected members of another class. It is named in the declaration of the other class with the prefix friend.

friend function. A function that is granted access to the private and protected parts of a class. It is named in the declaration of the class with the prefix friend.

full-motion video. (1) Video playback at 30 frames per second on NTSC signals. (2) A digital video compression technique that operates in real time.

G

gain. The ability to change the audibility of the sound, such as during a fade in or fade out of music.

graphic attributes. Attributes that apply to graphic primitives. Examples are color, line type, and shading-pattern definition.

graphic primitive. A single item of drawn graphics, such as a line, arc, or graphics text string.

graphical user interface (GUI). Type of computer interface consisting of a visual metaphor of a real-world scene, often of a desktop.

graphics. A picture defined in terms of graphic primitives and graphic attributes.

GUI. Graphical user interface.

H

halftone. The reproduction of continuous-tone artwork, such as a photograph, by converting the image into dots of various sizes.

hash function. A function that determines which category, or bucket, to put an element in. A hash function is needed when implementing a hash table.

hash table. A data structure that divides all elements into (preferably) equal-sized categories, or buckets, to allow quick access to the elements. The hash function determines which bucket an element belongs in.

header file. A file that can contain system-defined control information or user data and generally consists of declarations.

heap. An unordered flat collection that allows duplicate elements.

height of a tree. The length of the longest path from the root to a leaf.

hit testing. The means of identifying which graphic object the mouse is pointing to.

I

implementation class. A class that implements a concrete class. Implementation classes are never used directly.

incomplete class declaration. A class declaration that does not define any members of a class. Typically, you use an incomplete class declaration as a forward declaration.

indirection. A mechanism for connecting objects by storing, in one object, a reference to another object.

inheritance. (1) A mechanism by which a derived class can use the attributes, relationships, and member functions defined in more abstract classes related to it (its base classes). See also multiple inheritance. (2) An object-oriented programming technique that allows you to use existing classes as bases for creating other classes.

initializer. An expression used to initialize objects.

inlined function. A function call that the compiler replaces with the actual code for the function. You can direct the compiler to inline a function with the inline keyword.

input stream. A stream used to read input.

instance number. A number that the operating system uses to keep track of all of the instances of the same type of device. For example, the amplifier-mixer device name is AMPMIX plus a 2-digit instance number. If a program creates two amplifier-mixer objects, the device names could be AMPMIX01 and AMPMIX02.

integral object. A character object, an object having an enumeration type, an object having variations of the type int, or an object that is a bit field.

interactive graphics. Graphics that a user at a terminal can move or manipulate.

interactive video. The process of combining video and computer technology so that the user's actions, choices, and decisions affect the way in which the program unfolds.

interrupt. A temporary suspension of a process caused by an external event, performed in such a way that the process can be resumed.

intersection. Given collections A and B, the set of elements that is contained in both A and B.

intrinsic function. A function supplied by a program as opposed to a function supplied by the compiler.

inverted colors. Opposite colors in the light spectrum.

iteration. The process of repeatedly applying a function to a series of elements in a collection until some condition is satisfied.

iteration order. The order in which elements are accessed when iterating over a collection. In ordered collections, the element at position 1 will be accessed first, then the element at position 2, and so on. In sorted collections, the elements are accessed according to the ordering relation provided for the element type. In collections that are not ordered the elements are accessed in an arbitrary order. Each element is accessed exactly once.

iterator class. A class that provides iteration functions.

I/O Stream Library. A class library that provides the facilities to deal with many varieties of input and output.

K

key access. A property that allows elements to be accessed by matching keys.

key bag. An unordered flat collection that uses keys and can contain duplicate elements.

key collection. (1) An abstract class that has the property of key access. (2) In general, any collection that uses keys.

key equality. A relation that determines whether two keys are equal.

key() function. When used on a flat collection, a function that returns a reference to the key of an element.

key-type function. Any of several functions of an element type, that are used by the Collection Class Library member functions to manipulate the keys of a class.

key set. An unordered flat collection that uses keys and does not allow duplicate elements.

key sorted bag. A sorted flat collection that uses keys and allows duplicate elements.

key sorted collection. An abstract class with the properties of key equality and sorted elements.

key sorted set. A sorted flat collection that uses keys and does not allow duplicate elements.

keyword. (1) A predefined word reserved for the C or C++ language that you cannot use as an identifier. (2) A symbol that identifies a parameter.

L

last element. The element accessed last in an iteration over a collection. Each collection has its own definition for last element. For example, the last element of a sorted set is the element with the largest value.

latched. The state of a button. A button in its latched state is held in its pressed position until the user clicks on it to release (unlatch) it.

leaves. In a tree, nodes without children. Synonymous with terminals.

library. (1) A collection of functions, function calls, subroutines, or other data. (2) A set of object modules that can be specified in a link command.

linkage editor. Synonym for linker.

linked implementation. An implementation in which each element contains a reference to the next element in the collection. Pointer chains are used to access elements in linked implementations. Linked implementations are also called linked list implementations.

linked sequence. A sequence that uses a linked implementation.

linker. A program that resolves cross-references between separately compiled object modules and then assigns final addresses to create a single executable program.

locale. The definition of the subset of a user's environment that depends on language and cultural conventions.

lvalue. An expression that represents an object that can be both examined and altered.

M

manipulator. A value that can be inserted into streams or extracted from streams to affect or query the behavior of the stream.

mask. A pattern of bits or characters that controls the keeping, deleting, or testing of portions of another pattern of bits or characters.

MBCS. See multibyte character set

member. Data, functions, or types contained in classes, structures, or unions.

member function. An operator or function that is declared as a member of a class. A member function has access to the private and protected data members and member functions of objects of its class.

message. A request from one object that the receiving object implement a method. Because data is encapsulated and not directly accessible, a message is the only way to send data from one object to another. Each message specifies the name of the receiving object, the method to be implemented, and any parameters the method needs for implementation.

method. Synonym for member function.

MIDI. Musical Instrument Digital Interface. A standard used in the music industry for interfacing digital musical instruments.

mix. (1) An attribute that determines how the foreground of a graphic primitive is combined with the existing color of graphics output. Also known as foreground mix. Contrast with background mix. (2) The combination of audio or video sources during postproduction.

mixer. A device used to simultaneously combine and blend several inputs into one or two outputs.

mode. A collection of attributes that specifies a file's type and its access permissions.

motion video. Video that displays real motion.

mount. (1) To place a data medium in a position to operate. (2) To make recording media accessible.

Moving Pictures Experts Group (MPEG). (1) A group that is working to establish a standard for compressing and storing motion video and animation in digital form. (2) The compression standard of video and audio data that is stored on mass media.

MPEG. Moving Pictures Experts Group.

multibyte character set (MBCS). A character set whose characters consist of more than 1 byte. Used in languages such as Japanese, Chinese, and Korean, where the 256 possible values of a single-byte character set are not sufficient to represent all possible characters.

multimedia. Computer-controlled presentations combining any of the following: text, graphics, animation, full-motion images, still video images, and sound.

multiple inheritance. (1) An object-oriented programming technique implemented in C++ through derivation, in which the derived class inherits members from more than one base class. (2) The structuring of inheritance relationships among classes so a derived class can use the attributes, relationships, and functions used by more than one base class.

See also inheritance and class lattice.

multitasking. A mode of operation that allows concurrent performance or interleaved execution of more than one task or program.

multithread. Pertaining to concurrent operation of more than one path of execution within a computer.

N

n-ary tree. A tree that has an upper limit, *n*, imposed on the number of children allowed for a node.

National Television Standard Committee (NTSC). (1) A committee that sets the standard for color television broadcasting and video in the United States (currently in use also in Japan). (2) The standard set by the NTSC committee (the NTSC standard).

native. The rendering mechanism and format (RMF) that best represents the object and is the best one for rendering.

For example, a native of Cincinnati understands the streets in the area better than someone who has just moved there. Therefore, a Cincinnati native can get from point A to point B quicker than a newcomer. Likewise, a native RMF can get the data transferred from point A to point B more efficiently than the additional RMFs. We can use additional RMFs when we cannot use the native, or optimal, approach.

nested class. A class defined within the scope of another class.

new. (1) A C++ keyword identifying a free storage allocation operator. (2) A C++ operator used to create class objects.

new-line character. A control character that causes the print or display position to move to the first position on the next line. This control character is represented by \n in the C language.

node. In a tree structure, a point at which subordinate items of data originate.

NTSC. National Television Standard Committee.

NTSC format. The specifications for color television as defined by the NTSC, which include: (a) 525 scan lines, (b) broadcast bandwidth of 4 megaHertz, (c) line frequency of 15.75 kiloHertz, (d) frame frequency of 30 frames per second, and (e) color subcarrier frequency of 3.58 megaHertz.

null character (\0). The ASCII or EBCDIC character with the hex value 00 (all bits turned off).

O

object. (1) A collection of data and member functions that operate on that data, which together represent a logical entity in the system. In object-oriented programming, objects are grouped into classes that share common data definitions and functions. Each object of the class is said to be an instance of the class. (2) Each object has the same properties, attributes, and member functions as other objects of the same class, though it has unique values has unique values assigned to assigned to its attributes.

object-oriented programming. A programming approach based on the concepts of data abstraction and inheritance. Unlike procedural programming techniques, object-oriented programming concentrates on what data objects comprise the problem and how they are manipulated, not on how something is accomplished.

operation class. A class that defines all required element and key operations required by a specific collection implementation.

operator function. An overloaded operator that is either a member of a class or that takes at least one argument that is a class type or a reference to a class type. See overloading.

optical reflective disc. An optical videodisc that is read by means of the reflection of a laser beam from the shiny surface on the disc.

ordered collection. (1) An abstract class that has the property of ordered elements. (2) In general, any collection that has its elements arranged so that there is always a first element, last element, next element, and previous element.

ordering relation. A property that determines how the elements are sorted. Ascending order is an example of an ordering relation.

overflow. A condition that occurs when a portion of the result of an operation exceeds the capacity of the intended unit of storage.

overloading. An object-oriented programming technique where one or more function declarations are specified for a single name in the same scope.

owner window. A window similar to a parent window, but it does not affect the behavior or appearance of the window. The owner coordinates the activity of a window.

P

pad. To fill unused positions in a field with data, usually 0's, 1's, or blanks.

parameter declaration. A description of a value that a function receives. A parameter declaration determines the storage class and the data type of the value.

parent node. A node to which one or more other nodes are subordinate.

parent window. A window that provides the child window information on how and where to draw it. The parent window also defines the relationship that the child window has with other windows in the system.

pause. To temporarily halt the medium. The halted visual should remain displayed but no audio should be played.

pel. The smallest area of a display screen capable of being addressed and switched between visible and invisible states. Synonym for pixel and picture element.

picture element. Synonym for pel.

pitch. The ability to change the key or keynote of the sound. For example, in music, the different pitches of people's voices are soprano, alto, tenor, baritone, and bass, arranged from the highest to lowest pitch.

pixel. Picture element. Synonym for pel.

pointer. A variable that holds the address of a data object or function.

pointer class. A class that implements pointers.

pointer to member. An operator used to access the address of nonstatic members of a class.

polymorphic function. A function that can be applied to objects of more than one data type. C++ implements polymorphic functions in two ways:

1. Overloaded functions (calls are resolved at compile time)
2. Virtual functions (calls are resolved at run time)

polymorphism. The technique of taking an abstract view of an object or function and using any concrete objects or arguments that are derived from this abstract view.

positioning property. The property of an element that is used to position the element in a collection. For example, the value of the key may be used as the positioning property.

precondition. A condition that a function requires to be true when it is called.

predicate function. A function that returns an IBoolean value of *true* or *false*. (IBoolean is an integer-represented Boolean type.)

preparation. Any activity that the source performs before rendering the data. For example, the drag item may require that the source create a secondary thread for the source rendering to take place in. The system remains responsive to users so that they can do other tasks.

preprocessor. A phase of the compiler that examines the source program for preprocessor statements, which are then executed, resulting in the alteration of the source program.

preroll. To prepare a device to begin a playback or recording function with minimal delay.

primitive. See graphic primitive.

primitive attribute. A specifiable characteristic of a graphic primitive. See graphic attributes.

priority queue. A queue that has a priority assigned to its elements. When accessing elements, the element with the highest priority is removed first. A priority queue has a largest-in, first-out behavior.

private. Pertaining to a class member that is accessible only to member functions and friends of that class.

process. A program running under OS/2, along with the resources associated with it (memory, threads, file system resources, and so on).

profiling. The process of generating a statistical analysis of a program that shows processor time and the percentage of program execution time used by each procedure in the program.

program. (1) One or more files containing a set of instructions conforming to a particular programming language syntax. (2) A self-contained, executable module. Multiple copies of the same program can be run in different processes.

property function. A function that is used to determine whether the element it is applied to has a given property or characteristic. A property function can be used, for example, to remove all elements with a given property.

protected. Pertaining to a class member that is only accessible to member functions and friends of that class, or to member functions and friends of classes derived from that class.

prototype. A function declaration or definition that includes both the return type of the function and the types of its arguments.

public. Pertaining to a class member that is accessible to all functions.

pure virtual function. A virtual function that has a function initializer of the form **= 0;**.

Q

queue. A sequence with restricted access in which elements can only be added at the back end (or bottom) and removed from the front end (or top). A queue is characterized by first-in, first-out behavior and chronological order.

R

reference class. A class that links a concrete class to an abstract class. Reference classes make polymorphism possible with the Collection Classes.

relation. An unordered flat collection class that uses keys, allows for duplicate elements, and has element equality.

renderer. An object that renders data using a particular mechanism, such as using files or shared memory. It contains definitions of supported rendering mechanisms and formats and types. Renderers are maintained positionally (1-based).

rendering. The transfer or re-creation of the dragged object from the source window to the target window.

rendering format. Identifies the actual format of the data being rendered in a direct manipulation operation.

rendering mechanism. Identifies the actual format of the data being rendered in a direct manipulation operation.

resource file. A file that contains data used by an application, such as text strings and icons.

returned element. An element returned by a function as the return value.

RGB. Red, green, blue. A method of processing color images according to their red, green, and blue color content.

RMFs. Rendering mechanisms and formats.

root. A node that has no parent. All other nodes of a tree are descendants of the root.

S

samples-per-second. The number of times per second that the audio card records data from the audio input. For example, 44 kiloHertz is CD quality; 22 kiloHertz is FM music quality; and 11 kiloHertz is voice quality.

SBCS. See single-byte character set

scan. To search backward and forward at high speed on a CD audio device. Scanning is analogous to fast forwarding.

scope. That part of a source program in which an object is defined and recognized.

scope operator (::). An operator that defines the scope for the argument on the right. If the left argument is blank, the scope is global; if the left argument is a class name, the scope is within that class. Also called a scope resolution operator.

scroll increment. The number by which the current value of the circular slider is incremented or decremented when a user presses one of the circular slider control buttons.

sequence. A sequentially ordered flat collection.

sequential collection. An abstract class with the property of sequentially ordered elements.

siblings. All the children of a node are said to be siblings of one another.

single-byte character set (SBCS). A set of characters in which each character is represented by a 1-byte code.

SMPTE time code. A frame-numbering system developed by SMPTE that assigns a number to each frame of video.

The 8-digit code is in the form HH:MM:SS:FF (hours, minutes, seconds, frame number). The numbers track elapsed hours, minutes, seconds, and frames from any chosen point.

sorted bag. A sorted flat collection that allows duplicate elements.

sorted collection. (1) An abstract class with the property of sorted elements. (2) In general, any collection with sorted elements.

sorted map. A sorted flat collection with key and element equality.

sorted relation. A sorted flat collection that uses keys, has element equality, and allows duplicate elements.

sorted set. A sorted flat collection with element equality.

sound track. Synonymous with audio track.

sprite. A small graphic that can be moved independently around the screen, producing animated effects.

stack. A data structure in which new elements are added to and removed from the top of the structure. A stack is characterized by Last-In-First-Out (LIFO) behavior.

standard error. An output stream usually intended to be used for diagnostic messages.

standard input. An input stream usually intended to be used for primary data input. Standard input comes from the keyboard unless redirection or piping is used, in which case standard input can be from a file or the output from another command.

standard output. An output stream usually intended to be used for primary data output. When programs are run interactively, standard output usually goes to the display unless redirection or piping is used, in which case standard output can go to a file or to another command.

step backward. In multimedia applications, to move the medium backward one frame or segment at a time.

step forward. In multimedia applications, to move the medium forward one frame or segment at a time.

step frame. A function of devices such as digital video and videodisc players that enables a user to move frame-by-frame in either direction.

stream. (1) A continuous stream of data elements being transmitted, or intended for transmission, in character or binary-digit form, using a defined format. (2) A file access object that allows access to an ordered sequence of characters, as described by the ISO C standard. A stream provides the additional services of user-selectable buffering and formatted input and output.

stream buffer. A stream buffer is a buffer between the ultimate consumer, ultimate producer, and the I/O Stream Library functions that format data. It is implemented in the I/O Stream Library by the streambuf class and the classes derived from streambuf.

string. A contiguous sequence of characters.

structure. A construct that contains an ordered group of data objects. Unlike an array, the data objects within a structure can have varied data types.

subclass. See derived class.

subscript. One or more expressions, each enclosed in brackets, that follow an array name. A subscript refers to an element in an array.

subtree. A tree structure created by arbitrarily denoting a node to be the root node in a tree. A subtree is always part of a whole tree.

superclass. See base class and abstract class.

superset. Given two sets A and B, A is a superset of B if and only if all elements of B are also elements of A. That is, A is a superset of B if B is a subset of A.

T

tabular implementation. An implementation that stores the location of elements in tables. Elements in a tabular implementation are accessed by using indices to arrays.

tabular sequence. A sequence that uses a tabular implementation.

template. A family of classes or functions where the code remains invariant but operates with variable types.

terminals. Synonym for *leaves*.

this. A C++ keyword that identifies a special type of pointer in a member function, one that references the class object with which the member function was invoked.

this collection. The collection to which a function is applied.

thread. A unit of execution within a process.

throw expression. An argument to the C++ exception being thrown.

time code. See SMPTE time code.

tool bar. The area under the title bar that displays the tools available.

transparency. Refers to when a selected color on a graphics screen is made transparent to allow the video behind it to become visible.

transparent color. (1) A clear color used to indicate the part of the bitmap that is not drawn for the bitmap. The area under the bitmap is not overpainted for areas of the bitmap that are set to the transparent color. (2) Video information is considered as being present on the video plane that is maintained behind the graphics plane. When an area on the graphics plane is painted with a transparent color, the video information in the video plane is made visible.

trap. An unprogrammed conditional jump to a specified address that is automatically activated by hardware. A recording is made of the location from which the jump occurred.

treble. (1) The upper half of the whole vocal or instrumental tonal range. (2) The higher portion of the audio frequency range in sound recording.

tree. A hierarchical collection of nodes that can have an arbitrary number of references to other nodes. A unique path connects every two nodes.

true and additional. The most accurate or most descriptive (primary) type of an object (true) and the other or secondary types (additional). For example, if the object is a text file, its true type is text; if the file was a C source code file, its true type is C code.

try block. A block in which a known C++ exception is passed to a handler.

typed implementation class. A class that implements a concrete class and provides an interface that is specific to a given element type. This interface allows the compiler to verify that, for example, integers cannot be added to a set of strings.

typeless implementation class. A class that implements a concrete class and provides an interface that is not specific to a given element type.

U

ultimate consumer. The target of data in an I/O operation. An ultimate consumer can be a file, a device, or an array of bytes in memory.

ultimate producer. The source of data in an I/O operation. An ultimate producer can be a file, a device, or an array of byes in memory.

unbounded collection. A collection that has no upper limit on the number of elements it can contain.

undefined cursor. A cursor that may or may not be valid, and that may or may not refer to a different element of the collection from the element it referred to before the function call that resulted in its becoming undefined. An undefined cursor may refer to no element of the collection, and still be a valid cursor.

underflow. (1) A condition that occurs when the result of an operation is less than the smallest possible nonzero number. (2) Synonym for arithmetic underflow, monadic operation.

union. (1) Structures that can contain different types of objects at different times. Only one of the member objects can be stored in a union at any time. (2) Given the sets A and B, all elements of A, B, or both A and B.

unique collection. A collection in which the value of an element only occurs once; that is, there are no duplicate elements.

unload. To eject the medium from the device.

unordered collection. A collection that has no order to its elements.

V

VCR. Videocassette recorder.

VGA. Video graphics adapater.

video. Pertaining to the portion of recorded information that can be seen.

video attributes. The standard video attributes are: brightness, contrast, freeze, hue, saturation, and sharpness.

video graphics adapter (VGA). A graphics controller for color displays. The pel resolution of the video graphics adapter is 4:4.

videocassette recorder (VCR). A device for recording or playing back videocassettes.

videodisc. A disc on which programs have been recorded for playback on a computer or a television set; a recording on a videodisc. The most common format in the United States and Japan is an NTSC signal recorded on the optical reflective format.

videodisc player. A device that provides video playback for prerecorded videodiscs.

virtual function. A function of a class that is declared with the keyword virtual. The implementation that is executed when you make a call to a virtual function depends on the type of the object for which it is called. This is determined at run time.

volatile. An attribute of a data object that indicates the object is changeable beyond the control or detection of the compiler. Any expression referring to a volatile object is evaluated immediately, for example, assignments.

volume. The intensity of sound. A volume of 0 is minimum volume. A volume of 100 is maximum volume.

W

white space. Space characters, tab characters, form feed characters, and new-line characters.

wide character. A character whose range of values can represent distinct codes for all members of the largest extended character set specified among the supporting locales.

Numerics

24-bit color. A digital standard that uses 24 bits of information to describe each color pixel, providing up to 16.7 million colors in one image (the highest digital standard currently available).

8-bit color. A digital standard that uses 8 bits of information to describe each color pixel, providing up to 256 colors in one image (the standard for VGA displays).

Special Characters

(::) (double colon). Scope operator. An operator that defines the scope for the argument on the right. If the left argument is blank, the scope is global; if the left argument is a class name, the scope is within that class. Also called a scope resolution operator.

Bibliography

This bibliography lists the publications that make up the IBM VisualAge C++ library and publications of related IBM products referenced in this book. The list of related publications is not exhaustive but should be adequate for most VisualAge C++ users.

The IBM VisualAge C++ Library

The following books are part of the IBM VisualAge C++ library.

- *Read Me First!*, S25H-6956
- *Welcome to VisualAge C++*, S25H-6957
- *User's Guide*, S25H-6961
- *Programming Guide*, S25H-6958
- *Visual Builder User's Guide*, S25H-6960
- *Visual Builder Parts Reference*, S25H-6967
- *Building Visual Builder Parts for Fun and Profit*, S25H-6968
- *Open Class Library User's Guide*, S25H-6962
- *Open Class Library Reference*, S25H-6965
- *Language Reference*, S25H-6963
- *C Library Reference*, S25H-6964

The IBM VisualAge C++ BookManager Library

The following documents are available in VisualAge C++ in BookManager format.

- *Read Me First!*, S25H-6956
- *Welcome to VisualAge C++*, S25H-6957
- *User's Guide*, S25H-6961
- *Programming Guide*, S25H-6958
- *Visual Builder User's Guide*, S25H-6960
- *Visual Builder Parts Reference*, S25H-6967
- *Building Visual Builder Parts for Fun and Profit*, S25H-6968
- *Open Class Library User's Guide*, S25H-6962
- *Open Class Library Reference*, S25H-6965
- *Language Reference*, S25H-6963
- *C Library Reference*, S25H-6964

C and C++ Related Publications

- *Portability Guide for IBM C*, SC09-1405
- *American National Standard for Information Systems / International Standards Organization — Programming Language C (ANSI/ISO 9899-1990[1992])*
- *Draft Proposed American National Standard for Information Systems — Programming Language C++ (X3J16/92-0060)*

IBM OS/2 2.1 Publications

The following books describe the OS/2 2.1 operating system and the Developer's Toolkit 2.1.

- *OS/2 2.1 Using the Operating System*, S61G-0703
- *OS/2 2.1 Installation Guide*, S61G-0704
- *OS/2 2.1 Quick Reference*, S61G-0713
- *OS/2 2.1 Command Reference*, S71G-4112
- *OS/2 2.1 Information and Planning Guide*, S61G-0913
- *OS/2 2.1 Keyboard and Codepages*, S71G-4113
- *OS/2 2.1 Bidirectional Support*, S71G-4114
- *OS/2 2.1 Book Catalog*, S61G-0706
- *Developer's Toolkit for OS/2 2.1: Getting Started*, S61G-1634

IBM OS/2 3.0 Publications

- *User's Guide to OS/2 Warp*, G25H-7196-01

The following books make up the OS/2 3.0 Technical Library (G25H-7116).

- *Control Program Programming Guide*, G25H-7101
- *Control Program Programming Reference*, G25H-7102
- *Presentation Manager Programming Guide - The Basics*, G25H-7103
- *Presentation Manager Programming Guide - Advanced Topics*, G25H-7104

- *Presentation Manager Programming Reference*, G25H-7105
- *Graphics Programming Interface Programming Guide*, G25H-7106
- *Graphics Programming Interface Programming Reference*, G25H-7107
- *Workplace Shell Programming Guide*, G25H-7108
- *Workplace Shell Programming Reference*, G25H-7109
- *Information Presentation Facility Programming Guide*, G25H-7110
- *OS/2 Tools Reference*, G25H-7111
- *Multimedia Application Programming Guide*, G25H-7112
- *Multimedia Subsystem Programming Guide*, G25H-7113
- *Multimedia Programming Reference*, G25H-7114
- *REXX User's Guide*, S10G-6269
- *REXX Reference*, S10G-6268

Multimedia Books

The following books are available as part of IBM Multimedia Presentation Manager/2 Version 1.1 (MMPM/2). The IBM User Interface Class Library multimedia classes encapsulate and extend many of the MMPM/2 functions.

- *The OS/2 Multimedia Advantage*, S71G-2220
- *Application Programming Guide*, S71G-2221
- *Programming Reference*, S71G-2222
- *Subsystem Development Guide*, S71G-2223
- *Guide to Multimedia User Interface Design*, S41G-2922

Other Books You Might Need

The following list contains the titles of IBM books that you might find helpful. These books are not part of the VisualAge C++ or OS/2 libraries.

BookManager READ/2 Publications

- *IBM BookManager READ/2: General Information*, GB35-0800
- *IBM BookManager READ/2: Getting Started and Quick Reference*, SX76-0146
- *IBM BookManager READ/2: Displaying Online Books*, SB35-0801
- *IBM BookManager READ/2: Installation*, GX76-0147

Non-IBM Publications

Many books have been written about the C++ language and related programming topics. The authors use varying approaches and emphasis. The following is a sample of some non-IBM C++ publications that are generally available. This sample is not an exhaustive list. IBM does not specifically recommend any of these books, and other C++ books may be available in your locality.

- *The Annotated C++ Reference Manual* by Margaret A. Ellis and Bjarne Stroustrup, Addison-Wesley Publishing Company.
- *C++ Primer* by Stanley B. Lippman, Addison-Wesley Publishing Company.
- *Object-Oriented Design with Applications* by Grady Booch, Benjamin/Cummings.
- *Object-Oriented Programming Using SOM and DSOM* by Christina Lau, Van Nostrand Reinhold.
- *OS/2 C++ Class Library: Power GUI Programming with C Set ++* by Kevin Leong, William Law, Robert Love, Hiroshi Tsuji, and Bruce Olson, Van Nostrand Reinhold.

Suggested Reading for Collection Classes

These books contain explanations of data structures that may help you understand the data structures in the Collection Classes:

- *Data Structures and Algorithms* by Aho, Hopcroft, and Ullman, Addison-Wesley Publishing Company.
- *The Art of Computer Programming, Vol. 3: Sorting and Searching*, D.E. Knuth, Addison-Wesley Publishing Company.
- *C++ Components and Algorithms* by Scott Robert Ladd, M&T Publishing Inc.
- *A Systematic Catalogue of Reusable Abstract Data Types* by Juergen Uhl and Hans Albrecht Schmit, Springer Varlag.

Index

Communicating Your Comments to IBM

IBM VisualAge C++ for OS/2
Open Class Library User's Guide

Version 3.0

Publication No. S25H-6962-00

If there is something you like—or dislike—about this book, please let us know. You can use one of the methods listed below to send your comments to IBM. If you want a reply, include your name, address, and telephone number. If you are communicating electronically, include the book title, publication number, page number, or topic you are commenting on.

The comments you send should only pertain to the information in this book and its presentation. To request additional publications or to ask questions or make comments about the functions of IBM products or systems, you should talk to your IBM representative or to your IBM authorized remarketer.

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you.

If you are mailing a readers' comment form (RCF) from a country other than the United States, you can give it to the local IBM branch office or IBM representative for postage-paid mailing.

- If you prefer to send comments by mail, use the RCF at the back of this book.
- If you prefer to send comments by FAX, use this number:
  - United States and Canada: 416-448-6161
  - Other countries: (+1)-416-448-6161
- If you prefer to send comments electronically, use the network ID listed below. Be sure to include your entire network address if you wish a reply.
  - Internet: torrcf@vnet.ibm.com
  - IBMLink: toribm(torrcf)
  - IBM/PROFS: torolab4(torrcf)
  - IBMMAIL: ibmmail(caibmwt9)

Readers' Comments — We'd Like to Hear from You

IBM VisualAge C++ for OS/2
Open Class Library User's Guide

Version 3.0

Publication No. S25H-6962-00

Overall, how satisfied are you with the information in this book?

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Overall satisfaction	□	□	□	□	□

How satisfied are you that the information in this book is:

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Accurate	□	□	□	□	□
Complete	□	□	□	□	□
Easy to find	□	□	□	□	□
Easy to understand	□	□	□	□	□
Well organized	□	□	□	□	□
Applicable to your tasks	□	□	□	□	□

Please tell us how we can improve this book:

Thank you for your responses. May we contact you? □ Yes □ No

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you.

Name

Company or Organization

Phone No.

Address

Readers' Comments — We'd Like to Hear from You
S25H-6962-00

Cut or Fold
Along Line

Fold and Tape **Please do not staple** Fold and Tape

PLACE
POSTAGE
STAMP
HERE

IBM Canada Ltd. Laboratory
Information Development
2G/345/1150/TOR
1150 EGLINTON AVENUE EAST
NORTH YORK ONTARIO CANADA M3C 1H7

Fold and Tape **Please do not staple** Fold and Tape

S25H-6962-00

Cut or Fold
Along Line

IBM®

Part Number: 25H6962
Program Number: 30H1664
30H1665
30H1666

Printed in U.S.A.

25H6962

S25H-6962-00