IBM VisualAge C++ for OS/2

S25H-6965-00

Open Class Library Reference
Volume I

Version 3.0

IBM VisualAge C++ for OS/2

S25H-6965-00

Open Class Library Reference Volume I

Version 3.0

Note!

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

First Edition (May 1995)

This edition applies to Version 3.0 of IBM VisualAge C++ for OS/2 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.

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. 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

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 ++	Common User Access
CUA	IBM
IBMLink	Open Class
OS/2	SAA
Systems Application Architecture	VisualAge
WorkFrame	

About This Book

This book describes the classes and class members of the C++ class libraries that are part of IBM Open Class Library, the comprehensive set of class libraries provided with VisualAge C++. The book covers the following IBM Open Class libraries:

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

Volume I of this document does not provide information on classes or functions of the User Interface Class Library. See Volumes II and III for descriptions of that library's classes or functions.

The book is divided into parts, with one or more parts for each of the class libraries listed above.

Who Should Use This Book

This book is intended for skilled C++ programmers who understand the concept of classes. Programmers who want to work with the Collection Class Library should 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 files 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 member functions to date and time objects
- Use Data Access Builder generated source code in conjunction with the Data Access Builder classes to access a DB2/2 relational database.

How to Use This Book

For introductory information on the class libraries, or information on how to use the class libraries, see the *Open Class Library User's Guide*. For detailed information on a particular class or member function, use this book. If you know what library a class is in, you can look at the table of contents entries for that library, and find the corresponding class. If you know what class within a library a given function is in, you can look at the table of contents entries for that library, find the class, and look for the member function within that class. If you do not know what library or class to look in, you can use the index.

Classes are organized alphabetically within each class library, except where classes with similar uses or characteristics are grouped 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 Class Library, all functions of flat collections are described in "Flat Collection Member Functions" on page 101 , 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.

Source Files for Collection Class Library Examples

The Collection Class Library examples contained in this book can get you started on particular collection classes. Source code and makefiles for these examples are located in `...\ibmclass\samples\iclcc`. If you want to understand the examples in more detail, you can read through the source files and header files. See Appendix A, "Header Files for Collection Class Library Coding Examples" on page 575 for a listing of the example header files.

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.

Motif This icon identifies information that applies only to Motif** versions of the Data Types and Exceptions classes.

PM This icon identifies information that applies only to Presentation Manager versions of the Data Types and Exceptions classes.

This icon identifies portability information that you should refer to when you are writing programs that use the Data Types and Exceptions classes and you want those programs to run on multiple platforms.

Related Documentation

See “Bibliography” on page 601 for a list of related books and suggested reading materials.

Related Books

Part 1. Complex Mathematics Library

complex Class

This chapter describes the member functions of the `complex` class, the class that provides you with the facilities to manipulate complex numbers.

Derivation `complex` does not derive from any class.

Header File `complex` is declared in `complex.h`

Members The following members are provided for `complex`:

Method	Page	Method	Page
Constructors	8	conj	13
operator +	9	cos	12
operator +=, -=, *=. /=	10	cosh	12
operator !=	10	exp	11
operator *	9	imag	14
operator - (negation)	9	log	11
operator - (subtraction)	9	norm	13
operator /	10	polar	14
operator >>	11	pow	12
operator <<	11	real	14
operator ==	10	sin	13
abs	13	sinh	13
arg	13	sqrt	12

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

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

Constant Name	Description
M_PI	π
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

Constructors for complex

There are two versions of the `complex` constructor:

```
complex();
complex(double r, double i=0.0);
```

If you declare a `complex` object without specifying any values for the real or imaginary part of the complex value, the constructor that takes no arguments is used and the complex value is initialized to (0, 0). For example, the following declaration gives the object `comp` the value (0, 0):

```
complex comp;
```

If you give either one or two values in your declaration, the constructor that takes two arguments is used. If you only give one value, the real part of the complex object is initialized to that value, and the imaginary part is initialized to 0.

For example, the following declaration gives the object `comp2` the value (3.14, 0):

```
complex comp2(3.14);
```

If you give two values in the declaration, the real part of the complex object is initialized to the first value and the imaginary part is initialized to the second value. For example, the following declaration gives the object `comp3` the value (3.14, 6.44):

```
complex comp3(3.14, 6.44);
```

There is no explicit `complex` destructor.

Initializing complex Arrays

You can use the `complex` constructor to initialize arrays of complex numbers. If the list of initial values is made up of complex values, each array element is initialized to the corresponding value in the list of initial values. If the list of initial values is not made up of complex values, the real parts of the array elements are initialized to these initial values and the imaginary parts of the array elements are initialized to 0. In the following example, the elements of array `b` are initialized to the values in the initial value list, but only the real parts of elements of array `a` are initialized to the values in the initial value list.

```
#include <complex.h>

void main() {
  complex a[3] = {1.0, 2.0, 3.0};
  complex b[3] = {complex(1.0, 1.0), complex(2.0, 2.0),
                  complex(3.0, 3.0)};
  cout << "Here is the first element of a: " << a[0] << endl;
  cout << "Here is the first element of b: " << b[0] << endl;
}
```

This example produces the following output:

```
Here is the first element of a: ( 1, 0)
Here is the first element of b: ( 1, 1)
```

Mathematical Operators for complex

The `complex` operators described in this section have the same precedence as the corresponding real operators.

Addition

`friend complex` **`operator+`**`(complex` *`x`*`, complex` *`y`*`);`

The addition operator returns the sum of *x* and *y*.

Subtraction

`friend complex` **`operator-`**`(complex` *`x`*`, complex` *`y`*`);`

The subtraction operator returns the difference between *x* and *y*.

Negation

`friend complex` **`operator-`**`(complex` *`x`*`);`

The negation operator returns (- *a*, - *b*) when its argument is (*a*, *b*).

Multiplication

`friend complex` **`operator*`**`(complex` *`x`*`, complex` *`y`*`);`

The multiplication operator returns the product of *x* and *y*.

complex Mathematical Operators

Division

```
friend complex operator/(complex x, complex y);
```

The division operator returns the quotient of *x* divided by *y*.

Equality

```
friend int operator==(complex x, complex y);
```

The equality operator "==" returns a nonzero value if *x* equals *y*. This operator tests for equality by testing that the two real components are equal and that the two imaginary components are equal.

Because both components are `double` values, the equality operator tests for an *exact* match between the two sets of values. If you want an equality operator that can test for an absolute difference within a certain tolerance between the two pairs of corresponding components, you can use a function such as the `isequal` function defined in "Equality and Inequality Operators Test for Absolute Equality" in the *Open Class Library User's Guide*.

Inequality

```
friend int operator!=(complex x, complex y);
```

The inequality operator "! =" returns a nonzero value if *x* does not equal *y*. This operator tests for inequality by testing that the two real components are not equal and that the two imaginary components are not equal.

Because both components are `double` values, the inequality operator returns false only when both the real and imaginary components of the two values are identical. If you want an inequality operator that can test for an absolute difference within a certain tolerance between the two pairs of corresponding components, you can use a function such as the `is_not_equal` function defined in "Equality and Inequality Operators Test for Absolute Equality" in the *Open Class Library User's Guide*.

Mathematical Assignment Operators

```
void operator+=(complex x);
void operator-=(complex x);
void operator*=(complex x);
void operator/=(complex x);
```

The following list describes the functions of the mathematical assignment operators:

- `x += y` assigns the value of `x + y` to *x*.
- `x -= y` assigns the value of `x - y` to *x*.
- `x *= y` assigns the value of `x * y` to *x*.
- `x /= y` assigns the value of `x / y` to *x*.

Note: The 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
y += z;               // correct method involves splitting
x = y;                // expression into separate statements
```

Input and Output Operators for complex

Input Operator

`istream& **operator**>>(istream& *is*, complex& *c*);`

The input (or extraction) operator >> takes complex value *c* from the stream *is* in the form (*a,b*). The parentheses and comma are mandatory delimiters for input when the imaginary part of the complex number being read is nonzero. Otherwise, they are optional. In both cases, white space is optional.

Output Operator

`ostream& **operator**<<(ostream& *os*, complex *c*);`

The output (or insertion) operator << writes complex value *c* to the stream *os* in the form (*a,b*).

Mathematical Functions for complex

exp

`friend complex **exp**(complex *x*);`

`exp()` returns the complex value equal to e^x where *x* is the argument. Table 2 on page 17 shows the values returned by the default error-handling procedure for `exp()`.

log

`friend complex **log**(complex *x*);`

`log()` returns the natural logarithm of the argument *x*. Table 2 on page 17 shows the values returned by the default error-handling procedure for `log()`.

complex Trigonometric Functions

pow

```
friend complex pow(double d, complex z);
friend complex pow(complex c, int i);
friend complex pow(complex c, double d);
friend complex pow(complex c, complex z);
```

`pow()` returns the complex value x^y, where *x* is the first argument and *y* is the second argument. `pow()` is overloaded four times. If *d* is a double value, *i* is an integer value, and *c* and *z* are complex values, then `pow()` can produce any of the following results:

- d^z
- c^i
- c^d
- c^z

sqrt

```
friend complex sqrt(complex x);
```

`sqrt()` returns the square root of its argument. If *c* and *d* are real values, then every complex number *(a,b),* where:

- $a = c^2 - d^2$
- $b = 2cd$

has two square roots:

- (*c*,*d*)
- (*-c*,*-d*)

`sqrt()` returns the square root that has a positive real part, that is, the square root that is contained in the first or fourth quadrants of the complex plane.

Trigonometric Functions for complex

cos

```
friend complex cos(complex x);
```

`cos()` returns the cosine of *x.*

cosh

```
friend complex cosh(complex x);
```

`cosh()` returns the hyperbolic cosine of *x*. Table 2 on page 17 shows the values returned by the default error-handling procedure for `cosh()`.

sin

```
friend complex sin(complex x);
```

`sin()` returns the sine of *x*.

sinh

```
friend complex sinh(complex x);
```

`sinh()` returns the hyperbolic sine of *x*. Table 2 on page 17 shows the values returned by the default error-handling procedure for `sinh()`.

Magnitude Functions for complex

abs

```
friend double abs(complex x);
```

`abs()` returns the absolute value or magnitude of its argument. The absolute value of a complex value (*a,b*) is the positive square root of a^2+b^2.

norm

```
friend double norm(complex x);
```

`norm()` returns the square of the magnitude of its argument. If the argument *x* is equal to the complex number (*a,b*), `norm()` returns the value a^2+b^2. `norm()` is faster than `abs()`, but it is more likely to cause overflow errors.

Conversion Functions for complex

You can use the conversion functions in the Complex Mathematics Library to convert between the polar and standard complex representations of a value and to extract the real and imaginary parts of a complex value.

arg

```
friend double arg(complex x);
```

`arg()` returns the angle (in radians) of the polar representation of its argument. If the argument *x* is equal to the complex number (*a,b*), the angle returned is the angle in radians on the complex plane between the real axis and the vector (*a,b*). The return value has a range of $-\pi$ to π. See Figure 4 in the *Open Class Library User's Guide* for an illustration of the polar representation of complex numbers.

conj

```
friend complex conj(complex x);
```

`conj()` returns the complex value equal to (*a,-b*) if the input argument *x* is equal to (*a,b*).

complex Conversion Functions

polar

```
friend complex polar(double a, double b= 0);
```

`polar()` returns the standard complex representation of the complex number that has a polar representation (*a*,*b*).

real

```
friend double real(const complex& x);
```

`real()` extracts the real part of the complex number *x*.

imag

```
friend double imag(const complex& x);
```

`imag()` extracts the imaginary part of the complex number *x*.

c_exception Class

Use the `c_exception` class to handle errors that are created by the functions and operations in the `complex` class.

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

Derivation `c_exception` is not derived from any other class.

Header File `c_exception` is declared in `complex.h`.

Members The following members are provided for `c_exception`:

Member	Page	Member	Page
Constructor	15	name	15
arg1	15	retval	16
arg2	15	type	16

Constructor for c_exception

```
c_exception(char *n, const complex& a1,
                     const complex& a2 = complex_zero);
```

The `c_exception` constructor creates a `c_exception` object with `name` member equal to *n*, `arg1` member equal to *a1*, and `arg2` member equal to *a2*.

Data Members of c_exception

arg1, arg2

```
complex arg1;
complex arg2;
```

`arg1` and `arg2` are the arguments with which the function that caused the error was called.

name

```
char *name;
```

`name` is a string that contains the name of the function where the error occurred.

Errors Handled by the Complex Library

retval `complex **retval**;`

`retval` is the value that the default definition if the error handling function `complex_error()` returns. You can make your own definition of `complex_error()` return a different value.

type `int **type**;`

`type` describes the type of error that has occurred. It can take the following values that are defined in the `complex.h` header file:

- `SING` argument singularity
- `OVERFLOW` overflow range error
- `UNDERFLOW` underflow range error

Errors Handled by the Complex Mathematics Library

complex_error

```
friend int complex_error(c_exception& ce);
```

`complex_error()` is invoked by member functions of the Complex Mathematics Library when errors are detected. The argument *ce* refers to the `c_exception` object that contains information about the error. You can define your own procedures for handling errors by defining a function called `complex_error()` with return type `int` and a single parameter of type `c_exception&`.

If you define your own `complex_error()` function and this function returns a nonzero value, no error message will be generated and the external variable `errno` will not be set. If this function returns zero, `errno` is given the value of one of the following constants:

- `ERANGE` if the result is too large or too small
- `EDOM` if there is a domain error within a mathematical function

These constants are defined in `errno.h`.

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

For example, if the source file containing your definition of `complex_error()` is `source1.cpp`, then you would invoke the compiler like this:

```
icc source1.cpp /B"/NOE"
```

Errors Handled by the Complex Library

Default Error-Handling Procedures

If you do not define your own `complex_error()`, the default error-handling procedures will be invoked when an error occurs. The results for a given input complex value (*a, b*) depend on the kind of error and the sign of the cosine and sine of *b*. The following table shows the return value of the default error-handling procedure and the value given to `errno` for each function with input equal to the complex value (*a, b*).

Notes:

The following symbols appear in this table:

1. NA - not applicable. The result of the error depends on the sign of the cosine and sine of b (the imaginary part of the argument) unless "NA" appears in the Cosine b or Sine b columns.
2. HUGE - the maximum double value. This value is defined in `math.h`.

Table 2 (Page 1 of 2). Results of the Default Error-Handling Procedures

Function	Error	Cosine b	Sine b	Return Value	errno
cosh	a too large	nonnegative	nonnegative	(+HUGE,+HUGE)	ERANGE
cosh	a too large	nonnegative	negative	(+HUGE,-HUGE)	ERANGE
cosh	a too small	nonnegative	nonnegative	(+HUGE,-HUGE)	ERANGE
cosh	a too small	nonnegative	negative	(+HUGE,+HUGE)	ERANGE
cosh	a too small	negative	nonnegative	(-HUGE,-HUGE)	ERANGE
cosh	a too small	negative	negative	(-HUGE,+HUGE)	ERANGE
cosh	b too large	negative	nonnegative	(-HUGE,+HUGE)	ERANGE
cosh	b too large	negative	negative	(-HUGE,-HUGE)	ERANGE
cosh	b too small	NA	NA	(0,0)	ERANGE
exp	a too large	positive	positive	(+HUGE,+HUGE)	ERANGE
exp	a too large	positive	nonpositive	(+HUGE,-HUGE)	ERANGE
exp	a too large	nonpositive	positive	(-HUGE,+HUGE)	ERANGE
exp	a too large	nonpositive	nonpositive	(-HUGE,-HUGE)	ERANGE
exp	a too small	NA	NA	(0,0)	ERANGE
exp	b too large	NA	NA	(0,0)	ERANGE
exp	b too small	NA	NA	(0,0)	ERANGE
log	a too large	positive	positive	(+HUGE,0)	See note
sinh	a too large	nonnegative	nonnegative	(+HUGE,+HUGE)	ERANGE
sinh	a too large	nonnegative	negative	(+HUGE,-HUGE)	ERANGE
sinh	a too large	negative	nonnegative	(-HUGE,+HUGE)	ERANGE
sinh	a too large	negative	negative	(-HUGE,-HUGE)	ERANGE
sinh	a too small	nonnegative	nonnegative	(-HUGE,+HUGE)	ERANGE

Errors Not Handled by the Complex Library

Table 2 (Page 2 of 2). Results of the Default Error-Handling Procedures

Function	Error	Cosine b	Sine b	Return Value	errno
sinh	a too small	nonnegative	negative	`(-HUGE,-HUGE)`	`ERANGE`
sinh	a too small	negative	nonnegative	`(+HUGE,+HUGE)`	`ERANGE`
sinh	a too small	negative	negative	`(+HUGE,-HUGE)`	`ERANGE`
sinh	b too large	NA	NA	`(0,0)`	`ERANGE`
sinh	b too small	NA	NA	`(0,0)`	`ERANGE`

Note: `errno` is set to `EDOM` when the error for `log()` is detected. The message is stored in `DDE4.MSG`. The message number in `DDE4.MSG` is 90. When this message is displayed by the C/C++ Runtime Library, it is changed to 5090. For information on binding this message, see "Binding Runtime Messages" in the *IBM VisualAge C++ for OS/2 User's Guide and Reference*.

Errors Not Handled by the Complex Mathematics Library

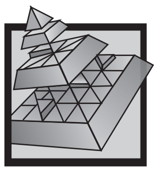
There are some cases where member functions of the Complex Mathematics Library call functions in the math library. These calls can cause underflow and overflow conditions that are handled by the `matherr()` function that is declared in the `math.h` header file. For example, the overflow conditions that 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. `INT_MAX` is the maximum `int` value. Both these constants are defined in `float.h`.

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

Part 2. I/O Stream Library



filebuf Class

This chapter describes the `filebuf` class, the class that specializes `streambuf` for using files as the ultimate producer or the ultimate consumer.

In a `filebuf` object, characters are cleared out of the put area by doing write operations to the file, and characters are put into the get area by doing read operations from that file. The `filebuf` class supports seek operations on files that allow seek operations. A `filebuf` object that is attached to a file descriptor is said to be open.

The stream buffer is allocated automatically if one is not specified explicitly with a constructor or a call to `setbuf()`. You can also create an unbuffered `filebuf` object by calling the constructor or `setbuf()` with the appropriate arguments. If the filebuf objec is unbuffered, a system call is made for each character that is read or written.

The get and put pointers for a `filebuf` object behave as a single pointer. This single pointer is referred to as the get/put pointer. The file that is attached to the `filebuf` object also has a single pointer that indicates the current position where information is being read or written. In this chapter, this pointer is called the *file get/put* pointer.

Derivation

streambuf
    filebuf

Header File

`filebuf` is declared in `fstream.h`.

Members

The following members are provided for `filebuf`:

Method	Page	Method	Page
filebuf constructor	22	is_open	23
filebuf destructor	22	open	23
attach	22	seekoff	23
close	22	seekpos	24
detach	22	setbuf	24
fd	23	sync	24

For an example of using the `filebuf` class, see “Using filebuf Functions to Move Through a File” in the *Open Class Library User's Guide*.

Public Members of filebuf

Note: The following descriptions assume that the functions are called as part of a `filebuf` object called *fb*.

Constructors for filebuf

```
filebuf();
filebuf(int d);
filebuf(int d, char* p, int len);
```

The `filebuf()` constructor with no arguments constructs an initially closed `filebuf` object.

The `filebuf()` constructor with one argument constructs a `filebuf` object that is attached to file descriptor *d*.

The `filebuf()` constructor with three arguments constructs a `filebuf` object that is attached to file descriptor *d*. The object is initialized to use the stream buffer starting at the position pointed to by *p* with length equal to *len*.

Destructor for filebuf

```
~filebuf();
```

The `filebuf` destructor calls *fb*`.close()`.

attach

```
filebuf* attach(int d);
```

`attach()` attaches *fb* to the file descriptor *d*. *fb* is the `filebuf` object returned by `attach()`. If *fb* is already open or if *d* is not open, `attach()` returns `NULL`. Otherwise, `attach()` returns a pointer to *fb*.

detach

```
int detach();
```

fb`.detach()` disconnects *fb* from the file without closing the file. If *fb* is not open, `detach()` returns -1. Otherwise, `detach()` flushes any output that is waiting in *fb* to be sent to the file, disconnects *fb* from the file, and returns the file descriptor.

close

```
filebuf* close();
```

`close()` does the following:

1. Flushes any output that is waiting in *fb* to be sent to the file
2. Disconnects *fb* from the file
3. Closes the file that was attached to *fb*

If an error occurs, `close()` returns 0. Otherwise, `close()` returns a pointer to *fb*. Even if an error occurs, `close()` performs the second and third steps listed above.

fd

```
int fd();
```

`fd()` returns the file descriptor that is attached to *fb*. If *fb* is closed, `fd()` returns `EOF`.

is_open

```
int is_open();
```

`is_open()` returns a nonzero value if *fb* is attached to a file descriptor. Otherwise, `is_open()` returns zero.

open

```
filebuf* open(const char* fname, int omode, int prot=openprot);
```

`open()` opens the file with the name *fname* and attaches *fb* to it. If *fname* does not already exist and *omode* does not equal `ios::nocreate`, `open()` tries to create it with protection mode equal to *prot*. The default value of *prot* is `filebuf::openprot`. An error occurs if *fb* is already open. If an error occurs, `open()` returns 0. Otherwise, `open()` returns a pointer to *fb*.

The default protection mode for the `filebuf` class is `S_IREAD | S_IWRITE`. If you create a file with both `S_IREAD` and `S_IWRITE` set, the file is created with both read and write permission. If you create a file with only `S_IREAD` set, the file is created with read-only permission, and cannot be deleted later with the `stdio.h` library function `remove()`. `S_IREAD` and `S_IWRITE` are defined in `sys\stat.h`.

seekoff

```
streampos seekoff(streamoff so, seek_dir sd, int omode);
```

`seekoff()` moves the file get/put pointer to the position specified by *sd* with the offset *so*. *sd* can have the following values:

- `ios::beg`: the beginning of the file
- `ios::cur`: the current position of the file get/put pointer
- `ios::end`: the end of the file

`seekoff()` changes the position of the file get/put pointer to the position specified by the value *sd* + *so*. The offset *so* can be either positive or negative. `seekoff()` ignores the value of *omode*.

If *fb* is attached to a file that does not support seeking, or if the value *sd* + *so* specifies a position before the beginning of the file, `seekoff()` returns `EOF` and the position of the file get/put pointer is undefined. Otherwise, `seekoff()` returns the new position of the file get/put pointer.

seekpos The `filebuf` class inherits the default definition of `seekpos()` from the `streambuf` class. The default definition defines `seekpos()` as a call to `seekoff()`. Thus, the following call to `seekpos()`:

```
seekpos(pos, mode);
```

is converted to a call to `seekoff()`:

```
seekoff(streamoff(pos), ios::beg, mode);
```

setbuf

```
streambuf* setbuf(char* pbegin, int len);
```

`setbuf()` sets up a stream buffer with length in bytes equal to *len*, beginning at the position pointed to by *pbegin*. `setbuf()` does the following:

- If *pbegin* is `0` or *len* is nonpositive, `setbuf()` makes *fb* unbuffered.
- If *fb* is open and a stream buffer has been allocated, no changes are made to this stream buffer, and `setbuf()` returns `NULL`.
- If neither of these cases is true, `setbuf()` returns a pointer to *fb*.

sync

```
int sync();
```

`sync()` attempts to synchronize the get/put pointer and the file get/put pointer. `sync()` may cause bytes that are waiting in the stream buffer to be written to the file, or it may reposition the file get/put pointer if characters that have been read from the file are waiting in the stream buffer. If it is not possible to synchronize the get/put pointer and the file get/put pointer, `sync()` returns `EOF`. If they can be synchronized, `sync()` returns zero.

fstream, ifstream, and ofstream Classes

The `fstream`, `ifstream`, and `ofstream` classes specialize `istream`, `ostream`, and `iostream` for use with files.

Derivation

- ios
  - istream
    - ifstream
  - ostream
    - ofstream
  - istream and ostream
    - iostream
      - fstream

Header File `fstream`, `ifstream`, and `ofstream` are declared in `fstream.h`.

Members The following members are provided for `fstream`, `ifstream`, `ofstream`, and `fstreambase`:

Method	Page	Method	Page
fstreambase:		**ifstream:**	
attach	26	constructor	28
close	26	open	29
detach	26	rdbuf	29
setbuf	26	**ofstream:**	
fstream:		constructor	29
constructor	26	open	30
open	27	rdbuf	30
rdbuf	28		

Public Members of fstreambase

Notes:

1. The `fstreambase` class is an internal class that provides common functions for the classes that are derived from it. Do not use the `fstreambase` class directly. The following descriptions are provided so that you can use the functions as part of `fstream`, `ifstream`, and `ofstream` objects.
2. The following descriptions assume that the functions are called as part of an `fstream`, `ifstream`, or `ofstream` object called *fb*.

attach

`void attach(int filedesc);`

`attach()` attaches *fb* to the file descriptor *filedesc*. If *fb* is already attached to a file descriptor, an error occurs and `ios::failbit` is set in the format state of *fb*.

close

`void close();`

`close()` closes the `filebuf` object, breaking the connection between *fb* and the file descriptor. `close()` calls *fb*`.rdbuf()->close()`. If this call fails, the error state of *fb* is not cleared.

detach

`int detach();`

`detach` detaches the `filebuf` object by calling *fb*`.rdbuf()->detach()`, and returns the value returned by *fb*`.rdbuf()->detach()`.

setbuf

`void setbuf(char* pbegin, int len);`

`setbuf()` sets up a stream buffer with length in bytes equal to *len* beginning at the position pointed to by *pbegin*. If *pbegin* is equal to 0 or *len* is nonpositive, *fb* will be unbuffered. If *fb* is open, or the call to *fb*`.rdbuf()->setbuf()` fails, `setbuf()` sets `ios::failbit` in the object's state.

Public Members of fstream

Note: The following descriptions assume that the functions are called as part of an `fstream` object called *fs*.

Constructors for fstream

`fstream();`

This version of the `fstream` constructor takes no arguments and constructs an unopened `fstream` object.

`fstream(int filedesc);`

This version takes one argument and constructs an `fstream` object that is attached to the file descriptor *filedesc*. If *filedesc* is not open, `ios::failbit` is set in the format state of *fs*.

`fstream(const char* fname, int mode, int prot=filebuf::openprot);`

This version constructs an `fstream` object and opens the file *fname* with open mode equal to *mode* and protection mode equal to *prot*. The default value for the argument `prot` is `filebuf::openprot`. If the file cannot be opened, the error state of the constructed `fstream` object is set.

```
fstream(int filedesc, char* bufpos, int len);
```

This version constructs an `fstream` object that is attached to the file descriptor *`filedesc`*. If *`filedesc`* is not open, `ios::failbit` is set in the format state of *`fs`*. This constructor also sets up an associated `filebuf` object with a stream buffer that has length *`len`* bytes and begins at the position pointed to by *`bufpos`*. If *`bufpos`* is equal to 0 or *`len`* is equal to 0, the associated `filebuf` object is unbuffered.

open

```
void open(const char* fname, int mode, int prot=filebuf::openprot);
```

`open()` opens the file with the name *`fname`* and attaches it to *`fs`*. If *`fname`* does not already exist, `open()` tries to create it with protection mode equal to *`prot`*, unless `ios::nocreate` is set.

The default value for `prot` is `filebuf::openprot`. If *`fs`* is already attached to a file or if the call to *`fs`*`.rdbuf()->open()` fails, `ios::failbit` is set in the error state for *`fs`*.

The members of the `ios::open_mode` enumeration are bits that can be `OR`ed together. The value of *`mode`* is the result of such an `OR` operation. This result is an **int** value, and for this reason, *`mode`* has type **int** rather than `open_mode`.

The elements of the `open_mode` enumeration have the following meanings:

ios::app	`open()` performs a seek to the end of the file. Data that is written is appended to the end of the file. This value implies that the file is open for output.
ios::ate	`open()` performs a seek to the end of the file. Setting `ios::ate` does not open the file for input or output. If you set `ios::ate`, you should explicitly set `ios::in`, `ios::out`, or both.
ios::bin	See `ios::binary` below.
ios::binary	The file is opened in binary mode. In the default (text) mode, carriage returns are discarded on input, as is an end-of-file (`0x1a`) character if it is the last character in the file. This means that a carriage return without an accompanying line feed causes the characters on either side of the carriage return to become adjacent. On output, a line feed is expanded to a carriage return and line feed. If you specify `ios::binary`, carriage returns and terminating end-of-file characters are not removed on input, and a line feed is not expanded to a carriage return and line feed on output. `ios::binary` and `ios::bin` provide identical functionality.

ios::in	The file is opened for input. If the file that is being opened for input does not exist, the open operation will fail. `ios::noreplace` is ignored if `ios::in` is set.
ios::out	The file is opened for output.
ios::trunc	If the file already exists, its contents will be discarded. If you specify `ios::out` and neither `ios::ate` nor `ios::app`, you are implicitly specifying `ios::trunc`. If you set `ios::trunc`, you should explicitly set `ios::in`, `ios::out`, or both.
ios::nocreate	If the file does not exist, the call to `open()` fails.
ios::noreplace	If the file already exists and `ios::out` is set, the call to `open()` fails. If `ios::out` is not set, `ios::noreplace` is ignored.

rdbuf

```
filebuf* rdbuf();
```

`rdbuf()` returns a pointer to the `filebuf` object that is attached to *`fs`*.

Public Members of ifstream

For an example of using the `ifstream` class, see "Opening a File for Input and Reading from the File" in the *Open Class Library User's Guide*.

Note: The following descriptions assume that the functions are called as part of an `ifstream` object called *`ifs`*.

Constructors for ifstream

```
ifstream();
```

This version of the `ifstream` constructor takes no arguments and constructs an unopened `ifstream` object.

```
ifstream(int filedesc);
```

This version takes one argument and constructs an `ifstream` object that is attached to the file descriptor *`filedesc`*. If *`filedesc`* is not open, `ios::failbit` is set in the format state of *`ifs`*.

```
ifstream(const char* fname,
        int mode=ios::in,
        int prot=filebuf::openprot);
```

The third version constructs an `ifstream` object and opens the file *`fname`* with open mode equal to *`mode`* and protection mode equal to *`prot`*. The default value for *`mode`* is `ios::in`, and the default value for *`prot`* is `filebuf::openprot`. If the file cannot be opened, the error state of the constructed `ifstream` object is set.

```
ifstream(int filedesc, char* bufpos, int len);
```

This version constructs an `ifstream` object that is attached to the file descriptor *`filedesc`*. If *`filedesc`* is not open, `ios::failbit` is set in the format state of *`ifs`*. This constructor also sets up an associated `filebuf` object with a stream buffer that has length *`len`* bytes and begins at the position pointed to by *`bufpos`*. If *`bufpos`* is equal to 0 or *`len`* is equal to 0, the associated `filebuf` object is unbuffered.

open

```
void open(const char* fname,
          int mode=ios::in,
          int prot=filebuf::openprot);
```

`open()` opens the file with the name *`fname`* and attaches it to *`ifs`*. If *`fname`* does not already exist, `open()` tries to create it with protection mode equal to *`prot`*, unless `ios::nocreate` is set in *`mode`*.

The default value for *`mode`* is `ios::in`. The default value for *`prot`* is `filebuf::openprot`. If *`ifs`* is already attached to a file, or if the call to *`ifs`*`.rdbuf()->open()` fails, `ios::failbit` is set in the error status for *`ifs`*.

The members of the `ios::open_mode` enumeration are bits that can be `OR`ed together. The value of *`mode`* is the result of such an `OR` operation. This result is an **int** value, and for this reason *`mode`* has type **int** rather than type `open_mode`. See "open" on page 27 for a list of the possible values for *`mode`*.

rdbuf

```
filebuf* rdbuf();
```

`rdbuf()` returns a pointer to the `filebuf` object that is attached to *`ifs`*.

Public Members of ofstream

For an example of using the `ofstream` class, see "Opening a File for Output and Writing to the File" in the *Open Class Library User's Guide*.

Note: The following descriptions assume that the functions are called as part of an `ofstream` object called *`ofs`*.

Constructors for ofstream

```
ofstream();
```

This version of the `ofstream` constructor takes no arguments and constructs an unopened `ofstream` object.

```
ofstream(int filedesc);
```

ofstream

This version takes one argument and constructs an `ofstream` object that is attached to the file descriptor *`filedesc`*. If *`filedesc`* is not open, `ios::failbit` is set in the format state of *`ofs`*.

```
ofstream(const char* fname,
        int mode=ios::out,
        int prot=filebuf::openprot);
```

This version constructs an `ofstream` object and opens the file *`fname`* with open mode equal to *`mode`* and protection mode equal to *`prot`*. The default value for *`mode`* is `ios::out`, and the default value for *`prot`* is `filebuf::openprot`. If the file cannot be opened, the error state of the constructed `ofstream` object is set.

```
ofstream(int filedesc, char* bufpos, int len);
```

This version constructs an `ofstream` object that is attached to the file descriptor *`filedesc`*. If *`filedesc`* is not open, `ios::failbit` is set in the format state of *`ofs`*. This constructor also sets up an associated `filebuf` object with a stream buffer that has length *`len`* bytes and begins at the position pointed to by *`bufpos`*. If *`p`* is equal to 0 or *`len`* is equal to 0, the associated `filebuf` object is unbuffered.

open

```
void open(const char* fname, int mode, int prot=filebuf::openprot);
```

`open()` opens the file with the name *`fname`* and attaches it to *`ofs`*. If *`fname`* does not already exist, `open()` tries to create it with protection mode equal to *`prot`*, unless `ios::nocreate` is set.

The default value for *`mode`* is `ios::out`. The default value for the argument *`prot`* is `filebuf::openprot`. If *`ofs`* is already attached to a file, or if the call to the function *`ofs`*`.rdbuf()->open()` fails, `ios::failbit` is set in the error state for *`ofs`*.

The members of the `ios::open_mode` enumeration are bits that can be `OR`ed together. The value of *`mode`* is the result of such an `OR` operation. This result is an **int** value, and for this reason, *`mode`* has type **int** rather than `open_mode`. See "open" on page 27 for a list of the possible values for *`mode`*.

rdbuf

```
filebuf* rdbuf();
```

`rdbuf()` returns a pointer to the `filebuf` object that is attached to *`ofs`*.

ios Class

The `ios` class maintains the error and format state information for the classes that are derived from it. The derived classes support the movement of formatted and unformatted data to and from the stream buffer. This chapter describes the members of the `ios` class, and thus describes the operations that are common to all the classes that are derived from `ios`.

Derivation ios

Header File `ios` is declared in `iostream.h`.

Members The following members are provided for `ios`. *Italicized* members are flags or variables used to maintain the format state information for streams.

Method	Page	Method	Page
ios constructor	32	precision	37
bad	40	pword	39
bitalloc	39	rdbuf	42
clear	40	rdstate	41
dec	34	*right*	34
dec manipulator	44	*scientific*	35
endl manipulator	44	setf	37
ends manipulator	44	*showbase*	34
eof	40	*showpoint*	35
fail	41	*showpos*	34
fill	36	skip	38
fixed	35	*skipws*	33
flags	37	*stdio*	36
flush manipulator	44	sync_with_stdio	42
good	41	tie	43
hex	34	*unitbuf*	36
hex manipulator	44	unsetf	38
internal	34	*uppercase*	35
iword	39	width	38
left	34	ws manipulator	44
oct	34	*x_fill*	32
oct manipulator	44	*x_precision*	33
operator void*	41	*x_width*	33
operator=	32	xalloc	39

Constructors and Assignment Operator for ios

```
public:
     ios(streambuf* sb);
protected:
     ios();
     init(streambuf* isb);
private:
     ios(ios& ioa);
     void operator=(ios& iob);
```

There are three versions of the `ios` constructor. The version that is declared `public` takes a single argument that is a pointer to the `streambuf` object that becomes associated with the constructed `ios` object. If this pointer is equal to 0, the result is undefined.

The version of the `ios` constructor that is declared `protected` takes no arguments. This version is needed because `ios` is used as a virtual base class for `iostream`, and therefore the `ios` class must have a constructor that takes no arguments. If you use this constructor in a derived class, you must use the `init()` function to associate the constructed `ios` object with the `streambuf` object pointed to by the argument *`isb`*.

Copying of `ios` objects is not well defined, and for this reason, both the assignment operator and the copy constructor are declared `private`. Assignment between streams is supported by the `istream_withassign`, `ostream_withassign`, and `iostream_withassign` classes. See "Assignment Operator for istream_withassign" on page 56 and "Assignment Operator for ostream_withassign" on page 68 for more details. Except for the `..._withassign` classes, none of the predefined classes derived from `ios` has a copy constructor or an assignment operator. Unless you define your own copy constructor or assignment operator for a class that you derive from `ios`, your class will have neither a copy constructor nor an assignment operator.

Format State Variables

The *format state* is a collection of *format flags* and *format variables* that control the details of formatting for input and output operations. This section describes the format variables.

x_fill

```
char x_fill;
```

`x_fill` is the character that is used to pad values that do not require the width of an entire field for their representation. Its default value is a space character.

x_precision `short x_precision;`

`x_precision` is the number of significant digits in the representation of floating-point values. Its default value is 6.

x_width `short x_width;`

`x_width` is the minimum width of a field. Its default value is 0.

Format State Flags

The following list shows the formatting features and the format flags that control them:

- White space 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`

“Mutually Exclusive Format Flags” on page 36 describes the flags that produce unpredictable results if they are set at the same time.

White Space and Padding

The following format state flags control white space and padding characters. `skipws` and `right` are set by default.

skipws If `ios::skipws` is set, white space will be skipped on input. If it is not set, white space is not skipped. If `ios::skipws` is not set, the arithmetic extractors will signal an error if you attempt to read an integer or floating-point value that is preceded by white space. `ios::failbit` is set, and extraction ceases until it is cleared. This is done to avoid looping problems. If the following program is run with an input file that contains integer values separated by spaces, `ios::failbit` is set after the first integer value is read, and the program halts. If the program did not call `fail()` at the beginning of the `while` loop to test if `ios::failbit` is set, it would loop indefinitely.

Format State Flags

```
#include <fstream.h>

void main()
{
   fstream f("spadina.dat", ios::in);
   f.unsetf(ios::skipws);
   int i;
   while (!f.eof() && !f.fail()) {
      f >> i;
      cout << i;
   }
}
```

left If `ios::left` is set, the value is left-justified. Fill characters are added after the value.

right If `ios::right` is set, the value is right-justified. Fill characters are added before the value.

internal If `ios::internal` is set, the fill characters are added after any leading sign or base notation, but before the value itself.

Base Conversion

The manipulators `ios::dec`, `ios::oct`, and `ios::hex` (see "Built-In Manipulators for ios" on page 44 for more details) have the same effect as the flags `ios::dec`, `ios::oct`, and `ios::hex`, respectively. `dec` is set by default.

dec If `ios::dec` is set, the conversion base is 10.

oct If `ios::oct` is set, the conversion base is 8.

hex If `ios::hex` is set, the conversion base is 16.

showbase If `ios::showbase` is set, the operation that inserts values converts them to an external form that can be read according to the C++ lexical conventions for integral constants. By default, `ios::showbase` is unset.

Integral Formatting

showpos If `ios::showpos` is set, the operation that inserts values places a positive sign "+" into decimal conversions of positive integral values. By default, `showpos` is not set.

Floating-Point Formatting

The following format flags control the formatting of floating-point values:

showpoint If `ios::showpoint` is set, trailing zeros and a decimal point appear in the result of a floating-point conversion. This flag has no effect if either `ios::scientific` or `ios::fixed` is set. `showpoint` is not set by default.

scientific If `ios::scientific` is set, the value is converted using scientific notation. In scientific notation, there is one digit before the decimal point and the number of digits following the decimal point depends on the value of `ios::x_precision`. The default value for `ios::x_precision` is 6. If `ios::uppercase` is set, an uppercase “E” precedes the exponent. Otherwise, a lowercase “e” precedes the exponent. By default, `uppercase` is not set. See “uppercase” for more information.

fixed If `ios::fixed` is set, floating-point values are converted to fixed notation with the number of digits after the decimal point equal to the value of `ios::x_precision` (or 6 by default). `ios::fixed` is not set by default.

Default Representation of Floating-Point Values

If neither `ios::fixed` nor `ios::scientific` is set, the representation of floating-point values depends on their values and the number of significant digits in the representation equals `ios::x_precision`. Floating-point values are converted to scientific notation if the exponent resulting from a conversion to scientific notation is less than -4 or greater than or equal to the value of `ios::x_precision`. Otherwise, floating-point values are converted to fixed notation. If `ios::showpoint` is not set, trailing zeros are removed from the result and a decimal point appears only if it is followed by a digit. `ios::scientific` and `ios::fixed` are collectively identified by the static member `ios::floatfield`.

Uppercase and Lowercase

The following enumeration member determines whether alphabetic characters used in floating-point numbers appear in upper- or lowercase:

uppercase If `ios::uppercase` is set, the operation that inserts values uses an uppercase “E” for floating-point values in scientific notation. In addition, the operation that inserts values stores hexadecimal digits “A” to “F” in uppercase and places an uppercase “X” before hexadecimal values when `ios::showbase` is set. If `ios::uppercase` is not set, a lowercase “e” introduces the exponent in floating-point values, hexadecimal digits “a” to “f” are stored in lowercase, and a lowercase “x” is inserted before hexadecimal values when `ios::showbase` is set.

The setting of `uppercase` also determines whether special numbers such as `inf` or `infinity` are inserted in uppercase.

Format State Members

Buffer Flushing

The following enumeration members affect buffer flushing behavior:

unitbuf If `ios::unitbuf` is set, `ostream::osfx()` performs a flush after each insertion. The attached stream buffer is *`unit buffered`*. `ios::unitbuf` is not set by default.

stdio This flag is used internally by `sync_with_stdio()`. Do not use `ios::stdio` directly. If you want to combine I/O Stream Library input and output with `stdio.h` input and output, use `sync_with_stdio()`. See "sync_with_stdio" on page 42 for more details on `sync_with_stdio()`. `ios::stdio` is not set by default.

Mutually Exclusive Format Flags

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`*. 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`

Public Members of ios for the Format State

You can use the member functions listed below to control the format state of an `ios` object.

Note: The following descriptions assume that the functions are called as part of an `ios` object called *`iosobj`*.

fill

```
char fill() const;
char fill(char fillchar);
```

`fill()` with no arguments returns the value of `ios::x_fill` in the format state of *`iosobj`*. `fill()` with an argument *`fillchar`* sets `ios::x_fill` to be equal to *`fillchar`*. It returns the value of `ios::x_fill`.

`ios::x_fill` is the character used as padding if the field is wider than the representation of a value. The default value for `ios::x_fill` is a space. The `ios::left`, `ios::right`, and `ios::internal` flags determine the position of the fill character. See "White Space and Padding" on page 33 for more details.

You can also use the parameterized manipulator `setfill` to set the value of `ios::x_fill`. See "setfill" on page 58 for a description of this parameterized manipulator.

Format State Members

flags

```
long flags() const;
long flags(long flagset);
```

`flags()` with no arguments returns the value of the flags that make up the current format state. `flags()` with one argument sets the flags in the format state to the settings specified in *`flagset`* and returns the value of the previous settings of the format flags.

precision

```
int precision() const;
int precision(int prec);
```

`precision()` with no arguments returns the value of `ios::x_precision`. `precision()` with one argument sets the value of `ios::x_precision` to *`prec`* and returns the previous value. The value of *`prec`* must be greater than 0. If the value is nonpositive, the value of `ios::x_precision` is set to the default value, 6. `ios::x_precision` controls the number of significant digits when floating-point values are inserted.

The format state in effect when `precision()` is called affects the behavior of `precision()`. If neither `ios::scientific` nor `ios::fixed` is set, `ios::x_precision` specifies the number of significant digits in the floating-point value that is being inserted. If, in addition, `ios::showpoint` is not set, all trailing zeros are removed and a decimal point only appears if it is followed by digits.

If either `ios::scientific` or `ios::fixed` is set, `ios::x_precision` specifies the number of digits following the decimal point.

You can also use the parameterized manipulator `setprecision` to set `ios::x_precision`. See "setprecision" on page 59 for more details on this parameterized manipulator.

setf

```
long setf(long newset);
long setf(long newset, long field);
```

`setf()` with one argument is accumulative. It sets the format flags that are marked in *`newset`*, without affecting flags that are *not* marked in *`newset`*, and returns the previous value of the format state. You can also use the parameterized manipulator `setiosflags` to set the format flags to a specific setting. See "setiosflags" on page 59 for more details on this parameterized manipulator.

`setf()` with two arguments clears the format flags specified in *`field`*, sets the format flags specified in *`newset`*, and returns the previous value of the format state. For example, to change the conversion base in the format state to `ios::hex`, you could use a statement like this:

```
s.setf(ios::hex, ios::basefield);
```

In this statement, `ios::basefield` specifies the conversion base as the format flag that is going to be changed, and `ios::hex` specifies the new value for the conversion base. If *newset* equals 0, all of the format flags specified in *field* are cleared. You can also use the parameterized manipulator `resetiosflags` to clear format flags. 🕮 See "resetiosflags" on page 58 for more details on this parameterized manipulator.

Note: If you set conflicting flags the results are unpredictable. 🕮 See "Mutually Exclusive Format Flags" on page 36 for more details.

skip

```
int skip(int i);
```

`skip()` sets the format flag `ios::skipws` if the value of the argument *i* does not equal 0. If *i* does equal 0, `ios::skipws` is cleared. `skip()` returns a value of 1 if `ios::skipws` was set prior to the call to `skip()`, and returns 0 otherwise.

unsetf

```
long unsetf(long oflags);
```

`unsetf()` turns off the format flags specified in *oflags* and returns the previous format state.

width

```
int width() const;
int width(int fwidth);
```

`width()` with no arguments returns the value of the current setting of the format state field width variable, `ios::x_width`. If the value of `ios::x_width` is smaller than the space needed for the representation of the value, the full value is still inserted.

`width()` with one argument, *fwidth*, sets `ios::x_width` to the value of *fwidth* and returns the previous value. The default field width is 0. When the value of `ios::x_width` is 0, the operations that insert values only insert the characters needed to represent a value.

If the value of `ios::x_width` is greater than 0, the characters needed to represent the value are inserted. Then fill characters are inserted, if necessary, so that the representation of the value takes up the entire field. `ios::x_width` only specifies a minimum width, not a maximum width. If the number of characters needed to represent a value is greater than the field width, none of the characters is truncated. After every insertion of a value of a numeric or string type (including `char*`, `unsigned char*`, `signed char*`, and `wchar_t*`, but excluding `char`, `unsigned char`, `signed char`, and `wchar_t`), the value of `ios::x_width` is reset to 0. After every extraction of a value of type `char*`, `unsigned char*`, `signed char*`, or `wchar_t*`, the value of `ios::x_width` is reset to 0.

You can also use the parameterized manipulator `setw` to set the field width. See "setw" on page 59 for more information on this parameterized manipulator. Also, see "Public Members of ostream for Formatted Output" on page 62 for more information on `ios::x_width`.

Public Members of ios for User-Defined Format Flags

In addition to the flags described in "Format State Flags" on page 33 , you can also use the `ios` member functions listed in this section to define additional format flags or variables in classes that you derive from `ios`.

bitalloc

`static long` **`bitalloc`**`();`

`bitalloc()` is a static function that returns a `long` value with a previously unallocated bit set. You can use this `long` value as an additional flag, and pass it as an argument to the format state member functions. When all the bits are exhausted, `bitalloc()` returns 0.

iword

`long&` **`iword`**`(int` *i*`);`

`iword()` returns a reference to the *i*th user-defined flag, where *i* is an index returned by `xalloc()`. `iword()` allocates space for the user-defined flag. If the allocation fails, `iword()` sets `ios::failbit`.

pword

`void* &` **`pword`**`(int` *i*`);`

`pword()` returns a reference to a pointer to the *i*th user-defined flag, where *i* is an index returned by `xalloc()`. `pword()` allocates space for the user-defined flag. If the allocation fails, `pword()` sets `ios::failbit`. `pword()` is the same as `iword()`, except that the two functions return different types.

xalloc

`static int` **`xalloc`**`();`

`xalloc()` is a static function that returns an unused index into an array of words available for use as format state variables by classes derived from `ios`.

`xalloc()` simply returns a new index; it does not do any allocation. `iword()` and `pword()` do the allocation, and if the allocation fails, they set `ios::failbit`. You should check `ios::failbit` after calling `iword()` or `pword()`.

Public Members of ios for the Error State

The error state is an enumeration that records the errors that take place in the processing of `ios` objects. It has the following declaration:

```
enum io_state { goodbit, eofbit, failbit, badbit, hardfail };
```

The error state is manipulated using the `ios` member functions described in this section.

Notes:

1. `hardfail` is a flag used internally by the I/O Stream Library. Do not use it.
2. The following descriptions assume that the functions are called as part of an `ios` object called *`iosobj`*.

bad

```
int bad() const;
```

`bad()` returns a nonzero value if `ios::badbit` is set in the error state of *`iosobj`*. Otherwise, it returns 0. `ios::badbit` is usually set when some operation on the `streambuf` object that is associated with the `ios` object has failed. It will probably not be possible to continue input and output operations on the `ios` object.

clear

```
void clear(int state=0);
```

`clear()` changes the error state of *`iosobj`* to *`state`*. If *`state`* equals 0 (its default), all of the bits in the error state are cleared. If you want to *set* one of the bits without clearing or setting the other bits in the error state, you can perform a bitwise `OR` between the bit you want to set and the current error state. For example, the following statement sets `ios::badbit` in *`iosobj`* and leaves all the other error state bits unchanged:

```
iosobj.clear(ios::badbit|iosobj.rdstate());
```

eof

```
int eof() const;
```

`eof()` returns a nonzero value if `ios::eofbit` is set in the error state of *`iosobj`*. Otherwise, it returns 0. `ios::eofbit` is usually set when an `EOF` has been encountered during an extraction operation.

ios Error State

fail

```
int fail() const;
```

`fail()` returns a nonzero value if either `ios::badbit` or `ios::failbit` is set in the error state. Otherwise, it returns 0.

good

```
int good() const;
```

`good()` returns a nonzero value if no bits are set in the error state of *iosobj*. Otherwise, it returns 0.

rdstate

```
int rdstate() const;
```

`rdstate()` returns the current value of the error state of *iosobj*.

operator void*

```
operator void*();
operator const void*() const;
```

The `void*` operator converts *iosobj* to a pointer so that it can be compared to 0. The conversion returns 0 if `ios::failbit` or `ios::badbit` is set in the error state of *iosobj*. Otherwise, a pointer value is returned. This value is not meant to be manipulated as a pointer; the purpose of the operator is to allow you to write statements such as the following:

```
if (cin)
     cout << "ios::badbit and ios::failbit are not set" << endl;
if (cin >> x)
     cout << "ios::badbit and ios::failbit are not set "
          << x << " was input" << endl;
```

operator!

```
int operator!() const;
```

The **!** operator returns a nonzero value if `ios::failbit` or `ios::badbit` is set in the error state of *iosobj*. You can use this operator to write statements like the following:

```
if (!cin)
   cout << "either ios::failbit or ios::badbit is set" << endl;
else
   cout << "neither ios::failbit nor ios::badbit is set"
        << endl;
```

Other Members of ios

This section describes the `ios` member functions that do not deal with the error state or the format state. These descriptions assume that the functions are called as part of an `ios` object called *`iosobj.`*

rdbuf

```
streambuf* rdbuf();
```

`rdbuf()` returns a pointer to the `streambuf` object that is associated with *`iosobj`*. This is the `streambuf` object that was passed as an argument to the `ios` constructor. See "Constructors and Assignment Operator for ios" on page 32 for more details on the `ios` constructor.

sync_with_stdio

```
static void sync_with_stdio();
```

`sync_with_stdio()` is a static function that solves the problems that occur when you call functions declared in `stdio.h` and I/O Stream Library functions in the same program. The first time that you call `sync_with_stdio()`, it attaches `stdiobuf` objects to the predefined streams **cin**, **cout**, and **cerr**. After that, input and output using these predefined streams can be mixed with input and output using the corresponding `FILE` objects (`stdin`, `stdout`, and `stderr`). This input and output are correctly synchronized.

If you switch between the I/O Stream Library formatted extraction functions and `stdio.h` functions, you may find that a byte is "lost." The reason is that the formatted extraction functions for integers and floating-point values keep extracting characters until a nondigit character is encountered. This nondigit character acts as a delimiter for the value that preceded it. Because it is not part of the value, `putback()` is called to return it to the stream buffer. If a C `stdio` library function, such as `getchar()`, performs the next input operation, it will begin input at the character after this nondigit character. Thus, this nondigit character is not part of the value extracted by the formatted extraction function, and it is not the character extracted by the C `stdio` library function. It is "lost." Therefore, you should avoid switching between the I/O Stream Library formatted extraction functions and C `stdio` library functions whenever possible.

`sync_with_stdio()` makes **cout** and **clog** unit buffered. See "Buffer Flushing" on page 36 for a definition of unit buffering. After you call `sync_with_stdio()`, the performance of your program could diminish. The performance of your program depends on the length of strings, with performance diminishing most when the strings are shortest.

Note: You should use I/O Stream Library functions exclusively for all new code.

tie

```
ostream* tie();
ostream* tie(ostream* os);
```

There are two versions of `tie()`. The version that takes no arguments returns the value of `ios::x_tie`, the tie variable. (The tie variable points to the `ostream` object that is tied to the `ios` object.) The version that takes one argument *`os`* makes the tie variable, `ios::x_tie`, equal to *`os`* and returns the previous value.

You can use `ios::x_tie` to automatically flush the stream buffer attached to an `ios` object. If `ios::x_tie` for an `ios` object is not equal to `0` and the `ios` object needs more characters or has characters to be consumed, the `ostream` object pointed to by `ios::x_tie` is flushed.

By default, the `tie` variables of the predefined streams **cin**, **cerr**, and **clog** all point to the predefined stream **cout**. The following example illustrates how these streams are tied:

```
// Tying two streams together
   #include <iostream.h>
   #include <fstream.h>

   void main() {
      float f;

      cout << "Enter a number: ";         // cin is tied to cout, so
      cin >> f;                           // cout is flushed before input
      cout << "The number was " << f << ".\n" << endl;

      ofstream myFile;
      myFile.open("testfile",ios::out);
      cin.tie(&myFile);                   // now tie cin to a different ostream

      cout << "Enter a number: ";         // cout is not flushed by cin,
      cin >> f;                           // so prompt appears after input.
      cout << "The number was " << f << ".\n" << endl;
   }
```

Initially, the program displays a prompt, requests input, and then displays output. After **cin** is tied to the `ofstream` `myFile`, however, the output is not flushed by the request for input, so no prompt is displayed until after the input is received. The output is flushed only by the `endl` manipulator at the end of the program. The following shows sample output for this program:

```
Enter a number: 5
The number was 5.

6
Enter a number: The number was 6.
```

Built-In Manipulators for ios

The I/O Stream Library provides you with a set of built-in manipulators for `ios` and the classes derived from it. These manipulators have a specific effect on a stream other than inserting or extracting a value. Manipulators implicitly invoke functions that modify the state of the stream, and they allow you to modify the state of a stream at the same time as you are doing input and output. The syntax for manipulators is consistent with the syntax for input and output.

The following is a list of the manipulators and the classes that they apply to:

`dec`	`istream` and `ostream`
`hex`	`istream` and `ostream`
`oct`	`istream` and `ostream`
`ws`	`istream`
`endl`	`ostream`
`ends`	`ostream`
`flush`	`ostream`

See “Built-In Manipulators for istream” on page 55 for more details on the built-in manipulators for `istream`. See “Built-In Manipulators for ostream” on page 67 for more details on the manipulators for `ostream`.

iostream and iostream_withassign Classes

The `iostream` class combines the input capabilities of the `istream` class with the output capabilities of the `ostream` class. It is the base class for three other classes that also provide both input and output capabilities:

- `iostream_withassign`, also described in this chapter, which you can use to assign another stream (such as an `fstream` for a file) to an `iostream` object.
- `strstream`, which is a stream of characters stored in memory.
- `fstream`, which is a stream that supports input and output.

Derivation

```
ios
    istream
    ostream
        iostream
            iostream_withassign
```

Header File `iostream` and `iostream_withassign` are declared in `iostream.h`.

Members The following members are provided for `iostream` and `iostream_withassign`:

Member	Page
iostream Constructor	45
iostream_withassign Constructor	45
iostream_withassign Assignment Operator	46

Public Members of iostream and iostream_withassign

Constructor for iostream

```
iostream(streambuf* sb);
```

The `iostream` constructor takes a single argument *sb*. The constructor creates an `iostream` object that is attached to the `streambuf` object that is pointed to by *sb*. The constructor also initializes the format variables to their defaults. See “Format State Variables” on page 32 for more details on the format variables.

Constructor for iostream_withassign

```
iostream_withassign();
```

The `iostream_withassign` constructor creates an `iostream_withassign` object. It does not do any initialization of this object.

Assignment Operator for iostream_withassign

```
iostream_withassign& operator=(ios& is);
iostream_withassign& operator=(streambuf* sb);
```

There are two versions of the `iostream_withassign` assignment operator. The first version takes a reference to an `ios` object, *is*, as its argument. It associates the stream buffer attached to *is* with the `iostream_withassign` object that is on the left side of the assignment operator.

The second version of the `iostream_withassign` assignment operator takes a pointer to a `streambuf` object, *sb*, as its argument. It associates this `streambuf` object with the `iostream_withassign` object that is on the left side of the assignment operator.

istream and istream_withassign Classes

This chapter describes the istream class and its derived class istream_withassign. You can use the istream member functions to take characters out of the stream buffer that is associated with an istream object. istream_withassign is derived from istream and includes an assignment operator.

Derivation

ios
  istream
    istream_withassign

Header File istream and istream_withassign are declared in iostream.h.

Members The following members are provided for istream and istream_withassign:

Method	Page	Method	Page
ipfx	48	tellg	54
istream Constructor	47	gcount	54
input operator	48	peek	55
get	52	putback	55
getline	53	sync	55
ignore	53	istream_withassign Constructor	56
read	54	istream_withassign operator=	56
seekg	54		

Constructors for istream

Constructor for istream

```
istream(streambuf* sb);
```

The istream constructor takes a single argument *sb*. The constructor creates an istream object that is attached to the streambuf object that is pointed to by *sb*. The constructor also initializes the format variables to their defaults. See “Format State Variables” on page 32 for details on the format variables.

The other istream constructor declarations in iostream.h are obsolete; do not use them.

Input Prefix Function

`int ipfx(int need=0);`

`ipfx()` checks the stream buffer attached to an `istream` object to determine if it is capable of satisfying requests for characters. It returns a nonzero value if the stream buffer is ready, and 0 if it is not.

The formatted input operator calls `ipfx(0)`, while the unformatted input functions call `ipfx(1)`.

If the error state of the `istream` object is nonzero, `ipfx()` returns 0. Otherwise, the stream buffer attached to the `istream` object is flushed if either of the following conditions is true:

- *need* has a value of 0.
- The number of characters available in the stream buffer is fewer than the value of *need*.

If `ios::skipws` is set in the format state of the `istream` object and *need* has a value of 0, leading white-space characters are extracted from the stream buffer and discarded. If `ios::hardfail` is set or `EOF` is encountered, `ipfx()` returns 0. Otherwise, it returns a nonzero value.

Public Members of istream for Formatted Input

You can use the `istream` class to perform formatted input from a stream buffer using the input operator >>. Consider the following statement, where *ins* is a reference to an `istream` object and *x* is a variable of a built-in type:

```
ins >> x;
```

The input operator >> calls `ipfx(0)`. If `ipfx()` returns a nonzero value, the input operator extracts characters from the `streambuf` object that is associated with *ins*. It converts these characters to the type of *x* and stores the result in *x*. The input operator sets `ios::failbit` if the characters extracted from the stream buffer cannot be converted to the type of *x*. If the attempt to extract characters fails because `EOF` is encountered, the input operator sets `ios::eofbit` and `ios::failbit`. If the attempt to extract characters fails for another reason, the input operator sets `ios::badbit`. Even if an error occurs, the input operator always returns *ins*.

The details of conversion depend on the format state (see "Format State Variables" on page 32 for details) of the `istream` object and the type of the variable *x*. The input operator may set the width variable `ios::x_width` to 0, but it does not change anything else in the format state. See "Input Operator for Arrays of Characters" on page 49 below for details.

Formatted Input

The input operator is defined for the following types:

- Arrays of character values (including `signed char` and `unsigned char`)
- Other integral values: `short`, `int`, `long`
- `float`, `double`, and `long double` values

In addition, the input operator is defined for `streambuf` objects.

You can also define input operators for your own types. For further details see "Defining an Input Operator for a Class Type" in the *Open Class Library User's Guide*.

The following sections describe the input operator for these types.

Note: The following descriptions assume that the input operator is called with the `istream` object *`ins`* on the left side of the operator.

Input Operator for Arrays of Characters

```
istream& operator>>(char* pc);
istream& operator>>(signed char* pc);
istream& operator>>(unsigned char* pc);
istream& operator>>(wchar_t* pwc);
```

For pointers to `char`, `signed char`, and `unsigned char`, the input operator stores characters from the stream buffer attached to *`ins`* in the array pointed to by *`pc`*. The input operator stores characters until a white-space character is found. This white-space character is left in the stream buffer, and the extraction stops. If `ios::x_width` does not equal zero, a maximum of `ios::x_width` - 1 characters are extracted. The input operator calls *`ins`*`.width(0)` to reset `ios::x_width` to 0.

For pointers to `wchar_t`, the input operator stores characters from the stream buffer attached to *`ins`* in the array pointed to by *`pwc`*. The input operator stores characters until a white-space character or a `wchar_t` blank is found. If the terminating character is a white-space character, it is left in the stream buffer. If it is a `wchar_t` blank, it is discarded to avoid returning two bytes to the input stream.

For `wchar_t*` arrays, if `ios::width` does not equal zero, a maximum of `ios::width-1` characters (at 2 bytes each) are extracted. A 2-character space is reserved for the `wchar_t` terminating null character.

Note: The input operators for these types also reset `ios::x_width` to 0. None of the other input operators affects `ios::x_width`. All of the output operators except those for the `char` types and `wchar_t`, on the other hand, reset `ios::x_width` to 0.

The input operator always stores a terminating null character in the array pointed to by *`pc`* or *`pwc`*, even if an error occurs. For arrays of `wchar_t*`, this terminating null character is a `wchar_t` terminating null character.

Formatted Input

Input Operator for char

```
istream& operator>>(char& rc);
istream& operator>>(signed char& rc);
istream& operator>>(unsigned char& rc);
istream& operator>>(wchar_t& rc);
```

For `char`, `signed char`, and `unsigned char`, the input operator extracts a character from the stream buffer attached to *ins* and stores it in *rc*.

For references to `wchar_t`, the input operator extracts a `wchar_t` character from the stream buffer and stores it in *wc*. If `ios::skipws` is set, the input operator skips leading `wchar_t` spaces as well as leading `char` white spaces.

Input Operator for Other Integral Values

```
istream& operator>>(short& ir);
istream& operator>>(unsigned short& ir);
istream& operator>>(int& ir);
istream& operator>>(unsigned int& ir);
istream& operator>>(long& ir);
istream& operator>>(unsigned long& ir);
```

This section describes how the input operator works for references to the integral types: `short`, `unsigned short`, `int`, `unsigned int`, `long`, and `unsigned long`. For these integral types, the input operator extracts characters from the stream buffer associated with *ins* and converts them according to the format state of *ins*. The converted characters are then stored in *ir*. There is no overflow detection on conversion of integral types.

The first character extracted from the stream buffer may be a sign (+ or -). The subsequent characters are converted until a nondigit character is encountered. This nondigit character is left in the stream buffer. Which characters are treated as digits depends on the setting of the following format flags:

- `ios::oct`: the characters are converted to an octal value. Characters are extracted from the stream buffer until a character that is not an octal digit (a digit from 0 to 7) is encountered. If `ios::oct` is set and a signed value is encountered, the value is converted into a decimal value. For example, if the characters "- 45" are encountered in the input stream and `ios::oct` is set, the decimal value - 37 is actually extracted.
- `ios::dec`: the characters are converted to a decimal value. Characters are extracted from the stream buffer until a character that is not a decimal digit (a digit from `0` to `9`) is encountered.
- `ios::hex`: the characters are converted to a hexadecimal value. Characters are extracted from the stream buffer until a character that is not a hexadecimal digit (a digit from 0 to 9 or a letter from "A" to "F", upper or lower case) is

encountered. If `ios::hex` is set and a signed value is encountered, the value is converted into a decimal value. For example, if the characters “-12” are encountered in the input stream and `ios::hex` is set, the decimal value -18 is actually extracted.

If none of these format flags is set, the characters are converted according to the C++ lexical conventions.
This conversion depends on the characters that follow the optional sign:

- If these characters are “0x” or “0X”, the subsequent characters are converted to a hexadecimal value.
- If the first character is “0” and the second character is not “x” or “X”, the subsequent characters are converted to an octal value.
- If neither of these cases is true, the characters are converted to a decimal value.

If no digits are available in the stream buffer (other than the “0” in “0X” or “0x” preceding a hexadecimal value), the input operator sets `ios::failbit` in the error state of *ins*.

Input Operator for float and double Values

```
istream& operator>>(float& ref);
istream& operator>>(double& ref);
istream& operator>>(long double& ref);
```

For `float`, `double`, and `long double` values, the input operator converts characters from the stream buffer attached to *ins* according to the C++ lexical conventions.

The following conversions occur for certain string values:

- If the value consists of the character strings “`inf`” or “`infinity`” in any combination of uppercase and lowercase letters, the string is converted to the appropriate type's representation of infinity.
- If the value consists of the character string “`nan`” in any combination of uppercase and lowercase letters, the string is converted to the appropriate type's representation of a NaN.

The resulting value is stored in *ref*. The input operator sets `ios::failbit` if no digits are available in the stream buffer or if the digits that are available do not begin a floating-point number.

Input Operator for streambuf Objects

```
istream& operator>>(streambuf* sb);
```

For pointers to `streambuf` objects, the input operator calls `ipfx(0)`. If `ipfx(0)` returns a nonzero value, the input operator extracts characters from the stream buffer attached to *ins* and inserts them in *sb*. Extraction stops when an `EOF` character is encountered. The input operator always returns *ins*.

Public Members of istream for Unformatted Input

You can use the functions listed in this section to extract characters from a stream buffer as a sequence of bytes. All of these functions call `ipfx(1)`. They only proceed with their processing if `ipfx(1)` returns a nonzero value. See “Input Prefix Function” on page 48 for more details on `ipfx()`.

Note: The following descriptions assume that the functions are called as part of an `istream` object called *ins*.

get

```
istream& get(char* ptr, int len, char delim='\n');
istream& get(signed char* ptr, int len, char delim='\n');
istream& get(unsigned char* ptr, int len, char delim='\n');
```

`get()` with three arguments extracts characters from the stream buffer attached to *ins* and stores them in the byte array beginning at the location pointed to by *ptr* and extending for *len* bytes. The default value of the *delim* argument is `'\n'`. Extraction stops when either of the following conditions is true:

- *delim* or `EOF` is encountered before *len-1* characters have been stored in the array. *delim* is left in the stream buffer and not stored in the array.
- *len-1* characters are extracted without *delim* or `EOF` being encountered.

`get()` always stores a terminating null character in the array, even if it does not extract any characters from the stream buffer. `get()` sets the `ios::failbit` if it encounters an `EOF` character before it stores any characters.

get

```
istream& get(streambuf& sb, char delim='\n');
```

`get()` with two arguments extracts characters from the stream buffer attached to *ins* and stores them in *sb*. The default value of the *delim* argument is “`\n`”. Extraction stops when any of the following conditions is true:

- An `EOF` character is encountered.
- An attempt to store a character in *sb* fails. `ios::failbit` is set in the error state of *ins*.
- *delim* is encountered. *delim* is left in the stream buffer attached to *ins*.

get

```
istream& get(char& cref);
istream& get(signed char& cref);
istream& get(unsigned char& cref);
istream& get(wchar_t& cref);
```

`get()` with a single argument extracts a single character or `wchar_t` from the stream buffer attached to *`ins`* and stores this character in *`cref`*.

get

```
int get();
```

`get()` with no arguments extracts a character from the stream buffer attached to *`ins`* and returns it. This version of `get()` returns `EOF` if `EOF` is extracted. `ios::failbit` is never set.

getline

```
istream& getline(char* ptr, int len, char delim='\n');
istream& getline(signed char* ptr, int len, char delim='\n');
istream& getline(unsigned char* ptr, int len, char delim='\n');
```

`getline()` extracts characters from the stream buffer attached to *`ins`* and stores them in the byte array beginning at the location pointed to by *`ptr`* and extending for *`len`* bytes. The default value of the *`delim`* argument is "`\n`". Extraction stops when any one of the following conditions is true:

- *`delim`* or `EOF` is encountered before *`len-1`* characters have been stored in the array. `getline()` extracts *`delim`* from the stream buffer, but it does not store *`delim`* in the array.
- *`len-1`* characters are extracted before delim or `EOF` is encountered.

`getline()` always stores a terminating null character in the array, even if it does not extract any characters from the stream buffer. `getline()` sets the `ios::failbit` for *`ins`* if it encounters an `EOF` character before it stores any characters.

`getline()` is like `get()` with three arguments, except that `get()` does not extract the *`delim`* character from the stream buffer, while `getline()` does.

See "White Space in String Input" in the *Open Class Library User's Guide* for an example of using the `getline()` function.

ignore

```
istream& ignore(int num=1, int delim=EOF);
```

`ignore()` extracts up to *`num`* character from the stream buffer attached to *`ins`* and discards them. `ignore()` will extract fewer than *`num`* characters if it encounters *`delim`* or `EOF`.

Positioning

read

```
istream& read(char* s, int n);
istream& read(signed char* s, int n);
istream& read(unsigned char* s, int n);
```

`read()` extracts *n* characters from the stream buffer attached to *ins* and stores them in an array beginning at the position pointed to by *s*. If `EOF` is encountered before `read()` extracts n characters, `read()` sets the `ios::failbit` in the error state of *ins*. You can determine the number of characters that `read()` extracted by calling `gcount()` immediately after the call to `read()`.

Public Members of istream for Positioning

seekg

```
istream& seekg(streampos sp);
istream& seekg(streamoff so, ios::seek_dir dir);
```

`seekg()` repositions the get pointer of the ultimate producer. `seekg()` with one argument sets the get pointer to the position *sp*. `seekg()` with two arguments sets the get pointer to the position specified by *dir* with the offset *so*. *dir* can have the following values:

- `ios::beg`: the beginning of the stream
- `ios::cur`: the current position of the get pointer
- `ios::end`: the end of the stream

If you attempt to set the get pointer to a position that is not valid, `seekg()` sets `ios::badbit`.

tellg

```
streampos tellg();
```

`tellg()` returns the current position of the get pointer of the ultimate producer.

Other Public Members of istream

Note: The following descriptions assume that the functions are called as part of an `istream` object called *ins*.

gcount

```
int gcount();
```

`gcount()` returns the number of characters extracted from the stream buffer attached to *ins* by the last call to an unformatted input function. (See “Public Members of istream for Unformatted Input” on page 52 for more details.) The input operator >> may call unformatted input functions, and thus formatted input may affect the value returned by `gcount()`. 🕮 See "Public Members of istream for Formatted Input" on page 48 for more details on formatted input.

peek

```
int peek();
```

`peek()` calls `ipfx(1)`. If `ipfx()` returns zero, or if no more input is available from the ultimate producer, `peek()` returns `EOF`. Otherwise, it returns the next character in the stream buffer attached to *ins* without extracting the character.

putback

```
istream& putback(char c);
```

`putback()` attempts to put a character that was extracted from the stream buffer attached to *ins* back into the stream buffer. *c* must equal the character before the get pointer of the stream buffer. Unless some other activity is modifying the stream buffer, this is the last character extracted from the stream buffer. If *c* is not equal to the character before the get pointer, the result of `putback()` is undefined, and the error state of *ins* may be set. `putback()` does not call `ipfx()`, but if the error state of *ins* is nonzero, `putback()` returns without putting back the character or setting the error state. 🕮 See "Input Prefix Function" on page 48 for more details on `ipfx()`.

sync

```
int sync();
```

`sync()` establishes consistency between the ultimate producer and the stream buffer attached to *ins*. `sync()` calls *ins*`.rdbuf()->sync()`, which is a virtual function, so the details of its operation depend on the way the function is defined in a given derived class. If an error occurs, `sync()` returns `EOF`.

Built-In Manipulators for istream

```
istream&     ws(istream&);
ios&         dec(ios&);
ios&         hex(ios&);
ios&         oct(ios&);
```

The I/O Stream Library provides you with a set of built-in manipulators that can be used with the `istream` class. These manipulators have a specific effect on an `istream` object beyond extracting their own values. The built-in manipulators are accepted by the following versions of the input operator:

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

If *ins* is a reference to an `istream` object, this statement extracts white-space characters from the stream buffer attached to *ins*:

```
ins >> ws;
```

This statement sets `ios::dec`:

```
ins >> dec;
```

This statement sets `ios::hex`:

```
ins >> hex;
```

This statement sets `ios::oct`:

```
ins >> oct;
```

Public Members of istream_withassign

Constructor for istream_withassign

```
istream_withassign();
```

The `istream_withassign` constructor creates an `istream_withassign` object. It does not do any initialization of this object.

Assignment Operator for istream_withassign

```
istream_withassign& operator=(istream& is);
istream_withassign& operator=(streambuf* sb);
```

There are two versions of the `istream_withassign` assignment operator. The first version takes a reference to an `istream` object, *is*, as its argument. It associates the stream buffer attached to *is* with the `istream_withassign` object that is on the left side of the assignment operator.

The second version of the assignment operator takes a pointer to a `streambuf` object, *sb*, as its argument. It associates this `streambuf` object with the `istream_withassign` object that is on the left side of the assignment operator.

Manipulators

This chapter describes the parameterized manipulators provided by the I/O Stream Library and the facilities you can use to declare your own manipulators.

Derivation The manipulator classes are defined by a set of macros, and take names as defined when you use the macros. 🕮 See Chapter 6, "Manipulators" on page 69 in the *Open Class Library User's Guide* for further information.

Header File The parameterized manipulator classes are declared in `iomanip.h`.

Members The following parameterized manipulators are described:

Manipulator	Page	Manipulator	Page
resetiosflags	58	setiosflags	59
setbase	58	setprecision	59
setfill	58	setw	59

Parameterized Manipulators for the Format State

The `iomanip.h` header file also contains calls to the `IOMANIPdeclare()` macro for types `int` and `long`. These calls create classes that are used to create the parameterized manipulators that control the format state of `ios` objects. 🕮 See "Format State Flags" on page 33 for a description of the format state.

The call to `IOMANIPdeclare(int)` creates classes with names that are expanded from the following macros:

- `SMANIP(int)`
- `SAPP(int)`
- `IMANIP(int)`
- `IAPP(int)`
- `OMANIP(int)`
- `OAPP(int)`
- `IOMANIP(int)`
- `IOAPP(int)`

All of these macros expand to names that include the string "int." Similarly, `IOMANIPdeclare(long)` creates eight classes whose names include the string "long."

Parameterized Manipulators for the Format State

The following manipulators are declared using the classes created by the calls to `IOMANIPdeclare(int)` and `IOMANIPdeclare(long)`.

Note: All of the parameterized manipulators described below are defined for both `istream` and `ostream` objects. In the following descriptions, *is* is a reference to an `istream` object and *os* is a reference to an `ostream` object.

resetiosflags `SMANIP(long)` **`resetiosflags`**`(long` *`flags`*`);`

`resetiosflags()` clears the format flags specified in *`flags`*. It can appear in an input stream:

```
is >> resetiosflags(flags);
```

In this case, `resetiosflags()` calls *`is`*`.setf(0,`*`flags`*`)`. See “setf” on page 37 for more details on `setf()`.

`resetiosflags()` can also appear in an output stream:

```
os << resetiosflags(flags);
```

In this case, `resetiosflags` calls *`os`*`.setf(0,`*`flags`*`)`.

setbase `SMANIP(int)` **`setbase`**`(int` *`base`*`);`

`setbase()` sets the conversion base to be equal to the value of the argument *`base`*. If *`base`* equals 10, the conversion base is set to 10. If *`base`* equals 8, the conversion base is set to 8. If *`base`* equals 16, the conversion base is set to 16. Otherwise, the conversion base is set to 0. If the conversion base is 0, output is treated the same as if the base were 10, but input is interpreted according to the C++ lexical conventions. This means that input values that begin with “0” are interpreted as octal values, and values that begin with “0x” or “0X” are interpreted as hexadecimal values.

setfill `SMANIP(int)` **`setfill`**`(int` *`fill`*`);`

`setfill()` sets the fill character, `ios::x_fill`, to *`fill`*. The fill character is the character that appears in values that need to be padded to fill the field width. `setfill()` can appear in either an input stream or an output stream:

```
is >> setfill(fill);
os << setfill(fill);
```

`setfill()` performs the same task as the function `fill()`. See “fill” on page 36 for more details on `fill()`.

Parameterized Manipulators for the Format State

setiosflags

`SMANIP(long) setiosflags(long flags);`

`setiosflags()` sets the format flags specified in *flags*. `setiosflags()` can appear in an input stream:

```
   is >> setiosflags(flags);
```

If it appears in an input stream, `setiosflags()` calls *is*`.setf.(`*flags*`)` See "setf" on page 37 for more details on `setf()`.

If it appears in an output stream, `setiosflags()` calls *os*`.setf(`*flags*`)`:

```
   os << setiosflags(flags);
```

setprecision

`SMANIP(int) setprecision(int prec);`

`setprecision()` sets the precision format state variable, `ios::x_prec`, to the value of *prec*. The value of *prec* must be greater than zero. If the value of *prec* is negative, the precision format state variable is set to 6. See "precision" on page 37 for a description of `ios::x_prec`.

`setprecision()` can appear in either an input stream or an output stream:

```
   is >> setprecision(prec);
   os << setprecision(prec);
```

setw

`SMANIP(int) setw(int width);`

`setw()` sets the width format state variable, `ios::x_width`, to the value of *width*. See "width" on page 38 for a description of what `ios::x_width` does.

`setw()` can appear in either an input stream or an output stream:

```
   is >> setw(width);
   os << setw(width);
```

Parameterized Manipulators for the Format State

ostream and ostream_withassign Classes

This chapter describes the `ostream` class and its derived class `ostream_withassign`. You can use the `ostream` member functions to put characters into the `streambuf` object that is associated with an `ostream` object. `ostream_withassign` is derived from `ostream` and includes an assignment operator.

Derivation

ios
  ostream
    ostream_withassign

Header File `ostream` and `ostream_withassign` are declared in `iostream.h`.

Members The following members are provided for `ostream` and `ostream_withassign`:

Method	Page	Method	Page
ostream constructors	61	osfx	62
output operator	63	put	66
ostream_withassign constructor	68	seekp	66
ostream_withassign operator=	68	tellp	67
flush	67	write	66
opfx	62		

Constructors for ostream

Constructor for ostream

```
ostream(streambuf* sb);
```

The `ostream` constructor takes a single argument, *sb*, which is a pointer to a `streambuf` object. The constructor creates an `ostream` object that is attached to the `streambuf` object pointed to by *sb*. The constructor also initializes the format variables to their defaults. See "Format State Variables" on page 32 for more details on the format variables.

The other declarations for the `ostream` constructor in `iostream.h` are obsolete; do not use them.

Output Prefix and Suffix Functions

The output operator calls the output prefix function `opfx()` before inserting characters into a stream buffer, and calls the output suffix function `osfx()` after inserting characters. The following descriptions assume the functions are called as part of an `ostream` object called *os*. See "Public Members of ostream for Formatted Output" for more details on formatted output.

opfx

`int **opfx**();`

`opfx()` is called by the output operator before inserting characters into a stream buffer. `opfx()` checks the error state of *os*. If the internal flag `ios::hardfail` is set, `opfx()` returns 0. Otherwise, `opfx()` flushes the stream buffer attached to the `ios` object pointed to by *os*`.tie()`, if one exists, and returns the value returned by `ios::good()`. `ios::good()` returns 0 if `ios::failbit`, `ios::badbit`, or `ios::eofbit` is set. Otherwise, `ios::good()` returns a nonzero value.

osfx

`void **osfx**();`

`osfx()` is called before a formatted output function returns. `osfx()` flushes the `streambuf` object attached to *os* if `ios::unitbuf` is set.

`osfx()` is called by the output operator. If you overload the output operator to handle your own classes, you should ensure that `osfx()` is called after any direct manipulation of a `streambuf` object. Binary output functions do not call `osfx()`.

Public Members of ostream for Formatted Output

The `ostream` class lets you use the output operator << to perform formatted output (or *insertion*) to a stream buffer. Consider the following statement, where *outs* is a reference to an `ostream` object and *x* is a variable of a built-in type:

```
outs << x;
```

The output operator << calls `opfx()` before beginning insertion. If `opfx()` (see "opfx") returns a nonzero value, the output operator converts *x* into a series of characters and inserts these characters into the stream buffer attached to *outs*. If an error occurs, the output operator sets `ios::failbit`.

The details of the conversion of *x* depend on the format state (see "Format State Flags" on page 33) of the `ostream` object and the type of *x*. For numeric and string values, including the `char*` types and `wchar_t*`, but excluding the `char` types and `wchar_t`, the output operator resets the width variable `ios::x_width` of the format state of an `ostream` object to 0, but it does not affect anything else in the format state.

Formatted Output

The output operator is defined for the following types:

- Arrays of characters and `char` values, including arrays of `wchar_t` and `wchar_t` values.
- Other integral values: `short`, `int`, `long`
- float, double and long double values
- Pointers to `void`

The following sections describe the output operators for these types. The output operator is also defined for `streambuf` objects.

You can also define output operators for your own types. 🕮 See "Defining an Output Operator for a Class Type" in the *Open Class Library User's Guide* for instructions on how to do this.

Note: The following descriptions assume that the output operator is called with the `ostream` object *`outs`* on the left side of the operator.

Output Operator for Arrays of Characters and char Values

```
ostream& operator<<(const char* cp);
ostream& operator<<(const signed char* cp);
ostream& operator<<(const unsigned char* cp);
ostream& operator<<(wchar_t);
ostream& operator<<(char ch);
ostream& operator<<(signed char ch);
ostream& operator<<(unsigned char ch);
ostream& operator<<(const wchar_t *);
```

For a pointer to a `char`, `signed char`, or `unsigned char` value, the output operator inserts all the characters in the string into the stream buffer with the exception of the null character that terminates the string. For a pointer to a `wchar_t`, the output operator converts the `wchar_t` string to its equivalent multibyte character string, and then inserts it into the stream buffer except for the null character that terminates the string.

If `ios::x_width` is greater than zero and the representation of the value to be inserted is less than `ios::x_width`, the output operator inserts enough fill characters to ensure that the representation occupies an entire field in the stream buffer.

The output operator does not perform any conversion on `char`, `signed char`, `unsigned char`, or `wchar_t` values.

Output Operator for Other Integral Values

```
ostream& operator<<(short iv);
ostream& operator<<(unsigned short iv);
ostream& operator<<(int iv);
ostream& operator<<(unsigned int iv);
ostream& operator<<(long iv);
ostream& operator<<(unsigned long iv);
ostream& operator<<(long long iv);
ostream& operator<<(unsigned long long iv);
```

Note: The last two operators above are only available when the compiler is in a mode that supports the `long long` data type.

For the integral types (`short`, `unsigned short`, `int`, `unsigned int`, `long`, and `unsigned long`), the output operator converts the integral value *iv* according to the format state of *outs* and inserts characters into the stream buffer associated with *outs.* There is no overflow detection on conversion of integral types.

The conversion that takes place on *iv* depends, in part, on the settings of the following format flags:

- If `ios::oct` is set, *iv* is converted to a series of octal digits. If `ios::showbase` is set, "0" is inserted into the stream buffer before the octal digits. If the value being inserted is equal to 0, a single "0" is inserted, not "00."
- If `ios::dec` is set, *iv* is converted to a series of decimal digits.
- If `ios::hex` is set, *iv* is converted to a series of hexadecimal digits. If `ios::showbase` is set, "0x" (or "0X" if `ios::uppercase` is set) is inserted into the stream buffer before the hexadecimal digits.

If none of these format flags is set, *iv* is converted to a series of decimal digits. If *iv* is converted to a series of decimal digits, its sign also affects the conversion:

- If *iv* is negative, a negative sign "-" is inserted before the decimal digits.
- If *iv* is equal to 0, the single digit 0 is inserted.
- If *iv* is positive and `ios::showpos` is set, a positive sign "+" is inserted before the decimal digits.

Output Operator for float and double Values

```
ostream& operator<<(float val);
ostream& operator<<(double val);
ostream& operator<<(long double val);
```

The output operator performs a conversion operation on the value *val* and inserts it into the stream buffer attached to *outs*. The conversion depends on the values returned by the following functions:

- *outs*`.precision()`: returns the number of significant digits that appear after the decimal. The default value is 6.

- *outs*.`width()`: if this returns 0, *val* is inserted without any fill characters. (📖 See “fill” on page 36 for more details on fill characters.) If the return value is greater than the number of characters needed to represent *val*, extra fill characters are inserted so that the total number of characters inserted is equal to the return value.

The conversion also depends on the values of the following format flags:

- If `ios::scientific` is set, *val* is converted to scientific notation, with one digit before the decimal, and the number of digits after the decimal equal to the value returned by *outs*.`precision()`. The exponent begins with a lowercase “e” unless `ios::uppercase` is set, in which case the exponent begins with an uppercase “E.”
- If `ios::fixed` is set, *val* is converted to fixed notation, with the number of digits after the decimal point equal to the value returned by *outs*.`precision()`. If neither `ios::fixed` nor `ios::scientific` is set, the conversion depends upon the value of *val*. 📖 See “Floating-Point Formatting” on page 35 for more details.
- If `ios::uppercase` is set, the exponents of values in scientific notation begin with an uppercase “E.”

📖 See “Format State Flags” on page 33 for more details on the format state.

Output Operator for Pointers to void

`ostream&` **`operator`**`<<(void*` *`vp`*`);`

The output operator converts pointers to `void` to integral values and then converts them to hexadecimal values as if `ios::showbase` were set. This version of the output operator is used to print out the values of pointers.

Output Operator for streambuf Objects

`ostream&` **`operator`**`<<(streambuf*` *`sb`*`);`

If `opfx()` returns a nonzero value, the output operator inserts all of the characters that can be taken from *sb* into the stream buffer attached to *outs*. Insertion stops when no more characters can be fetched from *sb*. No padding is performed. The return value is *outs*.

Public Members of ostream for Unformatted Output

You can use the functions listed in this section to insert characters into a stream buffer without regard to the type of the values that the characters represent.

Note: The following descriptions assume that the functions are called as part of an `ostream` object called *outs*.

put

```
ostream& put(char c);
```

`put()` inserts *c* in the stream buffer attached to *outs*. `put()` sets the error state of *outs* if the insertion fails.

write

```
ostream& write(const char* cp, int n);
ostream& write(const signed char* cp, int n);
ostream& write(const unsigned char* cp, int n);
```

`write()` inserts the *n* characters that begin at the position pointed to by *cp*. This array of characters does not need to end with a null character.

Public Members of ostream for Positioning

Note: The following descriptions assume that the functions are called as part of an `ostream` object called *outs*.

seekp

```
ostream& seekp(streampos sp);
ostream& seekp(streamoff so, ios::seek_dir dir);
```

`seekp()` repositions the put pointer of the ultimate consumer. `seekp()` with one argument sets the put pointer to the position *sp*. `seekp()` with two arguments sets the put pointer to the position specified by *dir* with the offset *so*. *dir* can have the following values:

- `ios::beg`: the beginning of the stream
- `ios::cur`: the current position of the put pointer
- `ios::end`: the end of the stream

The new position of the put pointer is equal to the position specified by *dir* offset by the value of *so*. If you attempt to move the put pointer to a position that is not valid, `seekp()` sets `ios::badbit`.

tellp

```
streampos tellp();
```

tellp() returns the current position of the put pointer of the stream buffer that is attached to *outs*.

Other Public Members of ostream

flush

```
ostream& flush();
```

The ultimate consumer of characters that are stored in a stream buffer may not necessarily consume them immediately. flush() causes any characters that are stored in the stream buffer attached to *outs* to be consumed. It calls *outs*.rdbuf()->sync() to accomplish this action.

Built-In Manipulators for ostream

```
ostream&    endl(ostream& i);
ostream&    ends(ostream& i);
ostream&    flush(ostream&);
ios&        dec(ios&);
ios&        hex(ios&);
ios&        oct(ios&);
```

The I/O Stream Library provides you with a set of built-in manipulators that can be used with the ostream class. These manipulators have a specific effect on an ostream object beyond extracting their own values. The built-in manipulators are accepted by the following versions of the output operators:

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

If *outs* is a reference to an ostream object, then this statement inserts a newline character and calls flush(). See “flush” for more details on flush().

```
outs << endl;
```

This statement inserts a null character:

```
outs << ends;
```

This statement flushes the stream buffer attached to *outs*. It is equivalent to flush()

```
outs << flush;
```

This statement sets ios::dec:

```
outs << dec;
```

This statement sets `ios::hex`:

```
   outs << hex;
```

This statement sets `ios::oct`:

```
   outs << oct;
```

Public Members of ostream_withassign

Constructor for ostream_withassign

```
ostream_withassign();
```

The `ostream_withassign` constructor creates an `ostream_withassign` object. It does not do any initialization on the object.

Assignment Operator for ostream_withassign

```
ostream_withassign& operator=(ostream& os);
ostream_withassign& operator=(streambuf* sb);
```

There are two versions of the `ostream_withassign` assignment operator. The first version takes a reference to an `ostream` object, *`os`*, as its argument. It associates the `streambuf` attached to *`os`* with the `ostream_withassign` object that is on the left side of the assignment operator.

The second version of the assignment operator takes a pointer to a `streambuf` object, *`sb`*, as its argument. It associates *`sb`* with the `ostream_withassign` object that is on the left side of the assignment operator.

stdiobuf and stdiostream Classes

This chapter describes the `stdiobuf` class and `stdiostream`, the class that uses `stdiobuf` objects as stream buffers. Operations on an `stdiobuf` are mirrored on the associated `FILE` structure (defined in the C header file `stdio.h`).

Note: The classes described in this chapter are meant to be used when you have to mix C code with C++ code. If you are writing new C++ code, use `filebuf`, `fstream`, `ifstream`, and `ofstream` instead of `stdiobuf` and `stdiostream`. See "fstream, ifstream, and ofstream Classes" on page 25 and "filebuf Class" on page 21 for more details on these classes. See "sync_with_stdio" on page 42 for information on synchronizing `stdio.h` input and output with I/O Stream Library input and output.

Derivation

ios
  stdiostream

streambuf
  stdiobuf

Header File

`stdiobuf` and `stdiostream` are declared in `stdiostr.h`.

Members

The following members are provided for `stdiobuf` and `stdiostream`:

Member	Page	Member	Page
stdiobuf		**stdiostream**	
Constructor	69	Constructor	70
Destructor	70	rdbuf	70
stdiofile	70		

Public Members of stdiobuf

Constructor for stdiobuf

```
stdiobuf(FILE* f);
```

The `stdiobuf` constructor creates an `stdiobuf` object that is associated with the `FILE` pointed to by *f*. Changes that are made to the stream buffer in an `stdiobuf` object are also made to the associated `FILE` pointed to by *f*.

Note: If `ios::stdio` is set in the format state of an `ostream` object, a call to `osfx()` flushes `stdout` and `stderr`.

Destructor for stdiobuf

~stdiobuf();

The stdiobuf destructor frees space allocated by the stdiobuf constructor and flushes the file that this stdiobuf object is associated with.

stdiofile FILE* **stdiofile**();

stdiofile() returns a pointer to the FILE object that the stdiobuf object is associated with.

Public Members of stdiostream

Constructor for stdiostream

stdiostream(FILE* *file*);

The stdiostream constructor creates a stdiostream object that is attached to the FILE pointed to by *file*.

rdbuf stdiobuf* **rdbuf**();

rdbuf() returns a pointer to the stdiobuf object that is attached to the stdiostream object.

Example of Using stdiostream The following example shows how you can use the stdiostream class. Two files are opened using fopen(). The pointers to the FILE structures are then used to create stdiostream objects. Finally, the contents of one of these stdiostream objects are copied into the other stdiostream object.

```
#include <stdiostr.h>
#include <stdio.h>
#include <stdlib.h>

void main()
{
   FILE *in = fopen("in.dat", "r");
   FILE *out = fopen("out.dat", "w");
   int c;
   if (in == NULL)
   {
        cerr << "Cannot open file 'in.dat' for reading."
             << endl;
        exit(1);
   }
   if (out == NULL)
   {
        cerr << "Cannot open file 'out.dat' for writing."
             << endl;
        exit(1);
   }
   //
```

```
      // Create a stdiostream object attached to "f"
      //
      stdiostream sin(in);
      stdiostream sout(out);
      cout << "The data contained in the file is: " << endl;
      //
      // Now read data from "sin" and copy it to
      // "cout" and "sout"
      //
      while ((c = sin.rdbuf()->sbumpc()) != EOF)
      {
         cout << char(c);
         sout.rdbuf()->sputc(c);
      }
      cout << endl;
   }
```

If you run this example with an input file containing the following:

```
   input input input input
```

The following output is produced:

```
The data contained in the file is:
   input input input input
```

stdiostream

streambuf Class

You can use the `streambuf` class to manipulate objects of its derived classes `filebuf`, `stdiobuf`, and `strstreambuf`, or to derive other classes from it.

Derivation `streambuf` is the base class for `strstream`, `stdiobuf`, and `filebuf`. It is not derived from any class.

Header File `streambuf` is declared in `iostream.h`.

Members The following members are provided for `streambuf`:

Method	Page	Method	Page
streambuf constructors	75	pptr	78
streambuf destructor	75	sbumpc	76
allocate	80	seekoff	83
base	77	seekpos	84
blen	80	setb	79
dbp	80	setbuf	84
doallocate	82	setg	79
eback	77	setp	79
ebuf	78	sgetc	76
egptr	78	sgetn	76
epptr	78	snextc	76
gbump	81	sputbackc	76
gptr	78	sputc	77
in_avail	75	sputn	77
out_waiting	76	stossc	77
overflow	82	sync	85
pbackfail	83	unbuffered	81
pbase	78	underflow	85
pbump	81		

streambuf Public and Protected Interfaces

`streambuf` has both a public interface and a protected interface. You should think of these two interfaces as being two separate classes, because the interfaces are used for different purposes. You should also treat `streambuf` as if it were defined as a virtual base class. Do not create objects of the `streambuf` class itself. This section describes the ways you can use the two interfaces of `streambuf`.

Although most virtual functions are declared `public`, you should overload them in the classes that you derive from `streambuf`, and consider them part of the protected interface.

What is the streambuf Public Interface?

You should not create objects of the `streambuf` public interface directly. Instead, you should use `streambuf` through one of the predefined classes derived from `streambuf`. You can use objects of `filebuf`, `strstreambuf` and `stdiobuf` (the predefined classes derived from `streambuf`) directly as implementations of stream buffers. The public interface consists of the `streambuf` public member functions that can be called on objects of these predefined classes. `streambuf` itself does not have any facilities for taking characters from the ultimate producer or sending them to the ultimate consumer. The specialized member functions that handle the interface with the ultimate producer and the ultimate consumer are defined in `filebuf`, `strstreambuf` and `stdiobuf`.

Except for the destructor of the `streambuf` class, the virtual functions are described as part of the protected interface.

What is the streambuf Protected Inteface?

Use the `streambuf` protected interface in the following ways:

- As a base class to implement your own specialized stream buffers. In this sense you can think of `streambuf` as a virtual base class. The `streambuf` class only provides the basic functions needed to manipulate characters in a stream buffer. The `filebuf`, `strstreambuf` and `stdiobuf` classes contain functions that handle the interface with the standard ultimate consumers and producers. If you want to perform more sophisticated operations, or if you want to use other ultimate consumers and ultimate producers, you will have to create your own class derived from `streambuf`. You need to know about the protected interface if you want to create a class derived from `streambuf`.
- Through a predefined class derived from `streambuf`.

There are two kinds of functions in the protected interface:

- Nonvirtual member functions, which manipulate `streambuf` objects at a level of detail that would be inappropriate in the public interface.
- Virtual member functions, which permit classes that you derive from `streambuf` to customize their operations depending on the ultimate producer or ultimate consumer. When you define the virtual functions in your derived classes, you must ensure that these definitions fulfill the conditions stated in the descriptions of the virtual functions. If your definitions of the virtual functions do not fulfill these conditions, objects of the derived class may have unspecified behavior. Although most virtual functions are declared as `public` members, they are

described with the protected interface (with the exception of the destructor for the `streambuf` class) because they are meant to be overridden in the classes that you derive from `streambuf`.

Public Members of the streambuf Public Interface

Note: The following descriptions assume that the functions are called as part of an object *fb* of a class derived from `streambuf`. *fb* could, for example, be an object of the class `filebuf`. It could also be an `strstreambuf` object or an `stdiobuf` object.

Constructors for streambuf

```
streambuf();
streambuf(char* buffer, int len);
streambuf(char* buffer, int len, int c); // obsolete
```

There are three versions of the constructor for `streambuf`. The version with no arguments constructs an empty stream buffer corresponding to an empty sequence. The values returned by `base()`, `eback()`, `ebuf()`, `egptr()`, `epptr()`, `pptr()`, `gptr()`, and `pbase()` are initially all zero for this stream buffer.

The version with two arguments constructs an empty stream buffer of length *len* starting at the position pointed to by *buffer*.

The version of the constructor with three arguments is obsolete. It is included in the I/O Stream Library for compatibility with the AT&T C++ Language System Release 1.2.

Destructor for streambuf

```
virtual ~streambuf();
```

The destructor for `streambuf` calls `sync()`. If a stream buffer has been set up and `ios::alloc` is set, `sync()` deletes the stream buffer. See “sync” on page 85 for more details on `sync()`.

in_avail

```
int in_avail();
```

`in_avail()` returns the number of characters that are available to be extracted from the get area of *fb*. You can extract the number of characters equal to the value that `in_avail()` returns without causing an error.

Public Members of streambuf

out_waiting

`int out_waiting();`

`out_waiting()` returns the number of characters that are in the put area waiting to be sent to the ultimate consumer.

sbumpc

`int sbumpc();`

`sbumpc()` moves the get pointer past one character and returns the character that it moved past. `sbumpc()` returns `EOF` if the get pointer is already at the end of the get area.

sgetc

`int sgetc();`

`sgetc()` returns the character after the get pointer without moving the get pointer itself. If no character is available, `sgetc()` returns `EOF`.

Note: `sgetc()` does not change the position of the get pointer.

sgetn

`int sgetn(char* ptr, int n);`

`sgetn()` extracts the *n* characters following the get pointer, and copies them to the area starting at the position pointed to by *ptr*. If there are fewer than *n* characters following the get pointer, `sgetn()` takes the characters that are available and stores them in the position pointed to by *ptr*. `sgetn()` repositions the get pointer following the extracted characters and returns the number of extracted characters.

snextc

`int snextc();`

`snextc()` moves the get pointer forward one character and returns the character following the new position of the get pointer. `snextc()` returns `EOF` if the get pointer is at the end of the get area either before or after it is moved forward.

sputbackc

`int sputbackc(char c);`

`sputbackc()` moves the get pointer back one character. The get pointer may simply move, or the ultimate producer may rearrange the internal data structures so that the character *c* is saved. The argument *c* must equal the character that precedes the get pointer in the stream buffer. The effect of `sputbackc()` is undefined if *c* is not equal to the character before the get pointer. `sputbackc()` returns `EOF` if an error occurs. The conditions that cause errors depend on the derived class.

sputc

```
int sputc(int c);
```

`sputc()` stores the argument *c* after the put pointer and moves the put pointer past the stored character. If there is enough space in the stream buffer, this will extend the size of the put area. `sputc()` returns `EOF` if an error occurs. The conditions that cause errors depend on the derived class.

sputn

```
int sputn(const char* s, int n);
```

`sputn()` stores the *n* characters starting at *s* after the put pointer and moves the put pointer to the end of the series. `sputn()` returns the number of characters successfully stored. If an error occurs, `sputn()` returns a value less than *n*.

stossc

```
void stossc();
```

`stossc()` moves the get pointer forward one character. If the get pointer is already at the end of the get area, `stossc()` does not move it.

Protected Functions That Return Pointers

This section describes the functions in the protected interface of `streambuf` that return pointers to boundaries of areas in a stream buffer.

Note: The following descriptions assume that the functions are called as part of an object called *dsb*, which is an object of a class that is derived from `streambuf`.

base

```
char* base();
```

`base()` returns a pointer to the first byte of the stream buffer. The stream buffer consists of the space between the pointer returned by `base()` and the pointer returned by `ebuf()`.

eback

```
char* eback();
```

`eback()` returns a pointer to the lower bound of the space available for the get area of *dsb*. The space between the pointer returned by `eback()` and the pointer returned by `gptr()` is available for *putback*. See “putback” on page 55 for details on *putback*.

Functions That Return Pointers

ebuf `char* ebuf();`

`ebuf()` returns a pointer to the byte after the last byte of the stream buffer.

egptr `char* egptr();`

`egptr()` returns a pointer to the byte after the last byte of the get area of *dsb*.

epptr `char* epptr();`

`epptr()` returns a pointer to the byte after the last byte of the put area of *dsb*.

gptr `char* gptr();`

`gptr()` returns a pointer to the first byte of the get area of *dsb*. The get area consists of the space between the pointer returned by `gptr()` and the pointer returned by `egptr()`. Characters are extracted from the stream buffer beginning at the character pointed to by `gptr()`.

pbase `char* pbase();`

`pbase()` returns a pointer to the beginning of the space available for the put area of *dsb*. Characters between the pointer returned by `pbase()` and the pointer returned by `pptr()` have been stored in the stream buffer, but they have not been consumed by the ultimate consumer.

pptr `char* pptr();`

`pptr()` returns a pointer to the beginning of the put area of *dsb*. The put area consists of the space between the pointer returned by `pptr()` and the pointer returned by `epptr()`.

Protected Functions That Set Pointers

This section describes the functions in the protected interface of `streambuf` that set the boundaries of areas in a stream buffer. The values of these boundaries are returned by the functions described in "Protected Functions That Return Pointers" on page 77.

Note: The following descriptions assume that the functions are called as part of an object called *dsb* which is an object of a class that is derived from `streambuf`.

setb

```
void setb(char* startbuf, char* endbuf, int delbuf = 0);
```

`setb()` sets the beginning of the existing stream buffer (the pointer returned by *dsb*`.base()`) to the position pointed to by *startbuf*, and sets the end of the stream buffer (the pointer returned by *dsb*`.ebuf()`) to the position pointed to by *endbuf*.

If *delbuf* is a nonzero value, the stream buffer will be deleted when `setb()` is called again. If *startbuf* and *endbuf* are both equal to 0, no stream buffer is established. If *startbuf* is not equal to 0, a stream buffer is established, even if *endbuf* is less than *startbuf*. If this is the case, the stream buffer has length zero.

setg

```
void setg(char* startpbk, char* startget, char* endget);
```

`setg()` sets the beginning of the get area of *dsb* (the pointer returned by *dsb*`.gptr()`) to *startget*, and sets the end of the get area (the pointer returned by *dsb*`.egptr()`) to *endget*. `setg()` also sets the beginning of the area available for putback (the pointer returned by *dsb*`.eback()`) to *startpbk*.

setp

```
void setp(char* startput, char* endput);
```

`setp()`sets the spaces available for the put area. Both the start (*pbase()*) and the beginning (*pptr()*) of the put area are set to the value *startput*. See Figure 6 on page 34 in the *Open Class Library User's Guide* for details on where each of these functions points to within the stream buffer.

`setp()` sets the beginning of the put area of *dsb* (the pointer returned by *dsb*`.pptr()`) to the position pointed to by *startput*, and sets the end of the put area (the pointer returned by *dsb*`.epptr()`) to the position pointed to by *endput*.

Other Nonvirtual Protected Member Functions

This section describes the remaining nonvirtual member functions that make up the protected interface of `streambuf`.

Note: The following descriptions assume that the functions are called as part of an object called *dsb* which is an object of a class that is derived from `streambuf`.

allocate

```
int allocate();
```

`allocate()` attempts to set up a stream buffer. `allocate()` returns the following values:

- 0, if *dsb* already has a stream buffer set up (that is, *dsb*`->base()` returns a nonzero value), or if `unbuffered()` returns a nonzero value. (See "unbuffered" on page 81 for more details.) `allocate()` does not do any further processing if it returns 0.
- 1, if `allocate()` does set up a stream buffer.
- `EOF`, if the attempt to allocate space for the stream buffer fails.

`allocate()` is not called by any other nonvirtual member function of `streambuf`.

blen

```
int blen() const;
```

`blen()` returns the length (in bytes) of the stream buffer.

dbp

```
void dbp();
```

`dbp()` writes to standard output the values returned by the following functions:

- `base()`
- `eback()`
- `ebuf()`
- `egptr()`
- `epptr()`
- `gptr()`
- `pptr()`

`dbp()` is intended for debugging. `streambuf` does not specify anything about the form of the output. `dbp()` is considered part of the protected interface because the information that it prints can only be understood in relation to that interface. It is declared as a public function so that it can be called anywhere during debugging.

The following example shows the output produced by `dbp()` when it is called as part of a `filebuf` object:

```
#include <iostream.h>
void main()
{
   cout << "Here is some sample output." << endl;
   cout.rdbuf()->dbp();
}
```

If you compile and run this example, your output will look like this:

```
Here is some sample output.
buf at 0x90210, base=0x91010, ebuf=0x91410,
pptr=0x91010, epptr=0x91410, eback=0, gptr=0, egptr=0
```

gbump

`void **gbump**(int *offset*);`

`gbump()` offsets the beginning of the get area by the value of *`offset`*. The value of *`offset`* can be positive or negative. `gbump()` does not check to see if the new value returned by `gptr()` is valid.

The beginning of the get area is equal to the value returned by `gptr()`. See "gptr" on page 78 for more details on `gptr()`.

pbump

`void **pbump**(int *offset*);`

`pbump()` offsets the beginning of the put area by the value of *`offset`*. The value of *`offset`* can be positive or negative. `pbump()` does not check to see if the new value returned by `pptr()` is valid.

The beginning of the put area is equal to the value returned by `pptr()`. See "pptr" on page 78 for more details on `pptr()`.

unbuffered

```
int unbuffered() const;
void unbuffered(int buffstate);
```

`unbuffered()` manipulates the private `streambuf` variable called the *buffering state*. If the buffering state is nonzero, a call to `allocate()` does not set up a stream buffer. See "allocate" on page 80 for more details on `allocate()`.

There are two versions of `unbuffered()`. The version that takes no arguments returns the current value of the buffering state. The version that takes an argument, *`buffstate`*, changes the value of the buffering state to *`buffstate`*.

Protected Virtual Member Functions

This section describes the virtual functions in the protected interface of `streambuf`. Although these virtual functions have default definitions in `streambuf`, they can be overridden in classes that are derived from `streambuf`. The following descriptions state the default definition of each function and the expected behavior for these functions in classes where they are overridden.

Note: The following descriptions assume that the functions are called as part of an object called *dsb*, which is an object of a class that is derived from `streambuf`.

doallocate

```
virtual int doallocate();
```

`doallocate()` is called when `allocate()` determines that space is needed for a stream buffer. See “allocate” on page 80 for more details on `allocate()`.

The default definition of `doallocate()` attempts to allocate space for a stream buffer using the operator `new`.

If you define your own version of `doallocate()`, it must call `setb()` to provide space for a stream buffer or return `EOF` if it cannot allocate space. `doallocate()` should only be called if `unbuffered()` and `base()` return zero.

In your own version of `doallocate()`, you provide the size of the buffer for your constructor. Assign the buffer size you want to to a variable using a `#define` statement. This variable can then be used in the constructor for your `doallocate()` function to define the size of the buffer. See “unbuffered” on page 81 for more details on `unbuffered()`. See “base” on page 77 for more details on `base()`.

overflow

```
virtual int overflow(int c = EOF);
```

`overflow()` is called when the put area is full, and an attempt is made to store another character in it. `overflow()` may be called at other times.

The default definition of `overflow()` is compatible with the AT&T C++ Language System Release 1.2 version of the stream package, but it is not considered part of the current I/O Stream Library. Thus, the default definition of `overflow()` should not be used, and every class derived from `streambuf` should define `overflow()` itself.

The definition of `overflow()` in your classes derived from `streambuf` should cause the ultimate consumer to consume the characters in the put area, call `setp()` to establish a new put area, and store the argument *c* in the put area if *c* does not equal `EOF`. `overflow()` should return `EOF` if an error occurs, and it should return a value not equal to `EOF` otherwise.

Virtual Member Functions

pbackfail

```
virtual int pbackfail(int c);
```

`pbackfail()` is called when both of the following conditions are true:

- An attempt has been made to put back a character.
- There is no room in the putback area. The pointer returned by `eback()` equals the pointer returned by `gptr()`. See "eback" on page 77 for more details on `eback()`. See "gptr" on page 78 for more details on `gptr()`.

The default definition of `pbackfail()` returns `EOF`.

If you define `pbackfail()` in your own classes, your definition of `pbackfail()` should attempt to deal with the full putback area by, for instance, repositioning the get pointer of the ultimate producer. If this is possible, `pbackfail()` should return the argument *c*. If not, `pbackfail()` should return `EOF`.

seekoff

```
virtual streampos seekoff(streamoff so, seek_dir dir,
         int mode = ios::in|ios::out);
```

`seekoff()` repositions the get or put pointer of the ultimate producer or ultimate consumer. `seekoff()` does not change the values returned by *dsb*.`gptr()` or *dsb*.`pptr()`.

The default definition of `seekoff()` returns `EOF`.

If you define your own `seekoff()` function, it should return `EOF` if the derived class does not support repositioning. If the class does support repositioning, `seekoff()` should return the new position of the affected pointer, or `EOF` if an error occurs. *so* is an offset from a position in the ultimate producer or ultimate consumer. *dir* is a position in the ultimate producer or ultimate consumer. *dir* can have the following values:

- `ios::beg`: the beginning of the ultimate producer or ultimate consumer
- `ios::cur`: the current position in the ultimate producer or ultimate consumer
- `ios::end`: the end of the ultimate producer or ultimate consumer

The new position of the affected pointer is the position specified by *dir* offset by the value of *so*. If you derive your own classes from `streambuf`, certain values of *dir* may not be valid depending on the nature of the ultimate consumer or producer.

If `ios::in` is set in *mode*, the `seekoff()` should modify the get pointer. If `ios::out` is set in *mode*, the put pointer should be modified. If both `ios::in` and `ios::out` are set, both the get pointer and the put pointer should be modified.

Virtual Member Functions

seekpos

```
virtual streampos seekpos(streampos pos,
          int mode = ios::in|ios::out);
```

seekpos() repositions the get or put pointer of the ultimate producer or ultimate consumer to the position *pos*. If ios::in is set in *mode*, the get pointer is repositioned. If ios::out is set in *mode*, the put pointer is repositioned. If both ios::in and ios::out are set, both the get pointer and the put pointer are affected. seekpos() does not change the values returned by *dsb*.gptr() or *dsb*.pptr().

The default definition of seekpos() returns the return value of the function seekoff(streamoff(*pos*), ios::beg, *mode*). Thus, if you want to define seeking operations in a class derived from streambuf, you can define seekoff() and use the default definition of seekpos().

If you define seekpos() in a class derived from streambuf, seekpos() should return EOF if the class does not support repositioning or if *pos* points to a position equal to or greater than the end of the stream. If not, seekpos() should return *pos*.

setbuf

```
virtual streambuf* setbuf(char* ptr, int len);
streambuf* setbuf(unsigned char* ptr, int len);
streambuf* setbuf(char* ptr, int len, int count); // obsolete
```

There are three versions of setbuf(). The two versions that take two arguments set up a stream buffer consisting of the array of bytes starting at *ptr* with length *len*.

This function is different from setb(). setb() sets pointers to an existing stream buffer. setbuf(), however, creates the stream buffer. The version of setbuf() that takes three arguments is obsolete. The I/O Stream Library includes it to be compatible with AT&T C++ Language System Release 1.2.

The default definition of setbuf() sets up the stream buffer if the streambuf object does not already have a stream buffer.

If you define setbuf() in a class derived from streambuf, setbuf() can either accept or ignore a request for an unbuffered streambuf object. The call to setbuf() is a request for an unbuffered streambuf object if *ptr* equals 0 or *len* equals 0. setbuf() should return a pointer to *sb* if it accepts the request, and 0 otherwise.

sync

`virtual int sync();`

`sync()` synchronizes the stream buffer with the ultimate producer or the ultimate consumer.

The default definition of `sync()` returns 0 if either of the following conditions is true:

- The get area is empty and there are no characters waiting to go to the ultimate consumer
- No stream buffer has been allocated for *sb*.

Otherwise, `sync()` returns `EOF`.

If you define `sync()` in a class derived from `streambuf`, it should send any characters that are stored in the put area to the ultimate consumer, and (if possible) send any characters that are waiting in the get area back to the ultimate producer. When `sync()` returns, both the put area and the get area should be empty. `sync()` should return `EOF` if an error occurs.

underflow

`virtual int underflow();`

`underflow()` takes characters from the ultimate producer and puts them in the get area.

The default definition of `underflow()` is compatible with the AT&T C++ Language System Release 1.2 version version of the stream package, but it is not considered part of the current I/O Stream Library. Thus, the default definition of `underflow()` should not be used, and every class derived from `streambuf` should define `underflow()` itself.

If you define `underflow()` in a class derived from `streambuf`, it should return the first character in the get area if the get area is not empty. If the get area is empty, `underflow()` should create a get area that is not empty and return the next character. If no more characters are available in the ultimate producer, `underflow()` should return `EOF` and leave the get area empty.

Virtual Member Functions

strstream, istrstream, and ostrstream Classes

This chapter describes `istrstream`, `ostrsteam`, and `strstream`, the classes that specialize `istream`, `ostream`, and `iostream` (respectively) to use `strstreambuf` objects for stream buffers. These classes are called the *array stream buffer* classes because their stream buffers are arrays of bytes in memory. You can use these classes to perform input and output with strings in memory.

This chapter also describes `strstreambase`, the class from which the array stream buffer classes are derived.

Derivation

```
ios
    istream
    ostream
        iostream
            strstream
ios
    istream
        istrstream
ios
    ostream
        ostrstream
```

Header File `strstream`, `istrstream`, and `ostrstream` are declared in `iostream.h`.

Members The following members are provided for `strstream`, `istrstream`, and `ostrstream`:

Method	Page	Method	Page
istrstream constructors	89	strstream destructor	88
istrstream destructor	89	pcount	90
ostrstream constructors	90	rdbuf	88
ostrstream destructor	90	str (strstream)	88
strstream constructor	88	str (ostrstream)	90

Public Members of strstreambase

Note: The `strstreambase` class is an internal class that provides common functions for the classes that are derived from it. Do not use the `strstreambase` class directly.

The following description is provided so that you can use the function as part of `istrstream`, `ostrsteam`, and `strstream` objects.

rdbuf

```
strstreambuf* rdbuf();
```

`rdbuf()` returns a pointer to the stream buffer that the `strstreambase` object is attached to.

Public Members of strstream

Constructor for strstream

```
strstream();
strstream(char* cp, int len, int mode);
strstream(signed char* cp, int len, int mode);
strstream(unsigned char* cp, int len, int mode);
```

There are two versions of the `strstream` constructor. The version that takes no arguments specifies that space is allocated dynamically for the stream buffer that is attached to the `strstream` object.

The version of the `strstream` constructor that takes three arguments specifies that characters should be extracted and inserted into the array of bytes that starts at the position pointed to by *cp* with a length of *len* bytes. If `ios::ate` or `ios::app` is set in *mode*, *cp* points to a null-terminated string and insertions begin at the null character. Otherwise, insertions begin at the position pointed to by *cp*. You can use the `istream::seekg()` function to reposition the get pointer anywhere in this array. See “seekg” on page 54 for more details on `seekg()`.

Destructor for strstream

```
~strstream();
```

The `strstream` destructor frees the space allocated by the `strstream` constructor.

str

```
char* str();
```

`str()` returns a pointer to the stream buffer attached to the `strstream` and calls `freeze()` (see “freeze” on page 93) with a nonzero value to prevent the stream buffer from being deleted. If the stream buffer was constructed with an explicit array, the value returned is a pointer to that array. If the stream buffer was constructed in dynamic mode, *cp* points to the dynamically allocated area.

Until you call `str()`, deleting the dynamically allocated stream buffer is the responsibility of the `strstream` object. After `str()` has been called, the calling application has responsibility for the dynamically allocated stream buffer.

If your application calls `str()` without calling `freeze()` with a nonzero argument (to unfreeze the `strstream`), or without explicitly deleting the array of characters returned by the call to `str()`, the array of characters will not be deallocated by the `strstream` when it is destroyed. This situation is a potential source of a memory leak.

Public Members of istrstream

Constructors for istrstream

```
istrstream(char* cp);
istrstream(signed char* cp);
istrstream(unsigned char* cp);
istrstream(const char* cp);
istrstream(const signed char* cp);
istrstream(const unsigned char* cp);
istrstream(char* cp, int len);
istrstream(signed char* cp, int len);
istrstream(unsigned char* cp, int len);
istrstream(const char* cp, int len);
istrstream(const signed char* cp, int len);
istrstream(const unsigned char* cp, int len);
```

The versions of the `istrstream` constructor that take one argument specify that characters should be extracted from the null-terminated string that is pointed to by *cp*. You can use the `istream::seekg()` function to reposition the get pointer in this string. See “seekg” on page 54 for more details on `seekg()`.

The versions of the `istrstream` constructor that take two arguments specify that characters should be extracted from the array of bytes that starts at the position pointed to by *cp* and has a length of *len* bytes. You can use `istream::seekg()` to reposition the get pointer anywhere in this array.

Destructor for istrstream

```
~istrstream();
```

The `istrstream` destructor frees space that was allocated by the `istrstream` constructor.

Public Members of ostrstream

Constructors for ostrstream

```
ostrstream();
ostrstream(char* cp, int len, int mode = ios::out);
ostrstream(signed char* cp, int len, int mode = ios::out);
ostrstream(unsigned char* cp, int len, int mode = ios::out);
```

The version of the `ostrsteam` constructor that takes no arguments specifies that space is allocated dynamically for the stream buffer that is attached to the `ostrsteam` object.

The versions of the `ostrsteam` constructor that take three arguments specify that the stream buffer that is attached to the `ostrsteam` object consists of an array that starts at the position pointed to by *`cp`* with a length of *`len`* bytes. If `ios::ate` or `ios::app` is set in *`mode`*, *`cp`* points to a null-terminated string and insertions begin at the null character. Otherwise, insertions begin at the position pointed to by *`cp`*. You can use the `ostream::seekp()` function to reposition the put pointer. See "seekg" on page 54 for more details on `seekg()`.

Destructor for ostrstream

```
~ostrstream();
```

The `ostrsteam` destructor frees space allocated by the `ostrsteam` constructor. The destructor also writes a null byte to the stream buffer to terminate the stream.

str

```
char* str();
```

`str()` returns a pointer to the stream buffer attached to the `ostrsteam` and calls `freeze()` (see "freeze" on page 93) with a nonzero value to prevent the stream buffer from being deleted. If the stream buffer was constructed with an explicit array, the value returned is a pointer to that array. If the stream buffer was constructed in dynamic mode, *`cp`* points to the dynamically allocated area.

Until you call `str()`, deleting the dynamically allocated stream buffer is the responsibility of the `ostrsteam` object. After `str()` has been called, the calling application has responsibility for the dynamically allocated stream buffer.

pcount

```
int pcount();
```

`pcount()` returns the number of bytes that have been stored in the stream buffer. `pcount()` is mainly useful when binary data has been stored and the stream buffer attached to the `ostrsteam` object is not a null-terminated string. `pcount()` returns the total number of bytes, not just the number of bytes up to the first null character.

strstreambuf Class

This chapter describes the `strstreambuf` class, the class that specializes `streambuf` to use an array of bytes in memory as the ultimate producer or ultimate consumer.

Derivation streambuf
    strstreambuf

Header File `strstreambuf` is declared in `strstream.h`.

Members The following members are provided for `strstreambuf`:

Method	Page	Method	Page
strstreambuf constructors	91	seekoff	93
strstreambuf destructors	92	setbuf	94
doallocate	93	str	93
freeze	93	underflow	94
overflow	93		

Public Members of strstreambuf

Constructors for strstreambuf

```
strstreambuf();
strstreambuf(int bufsize);
strstreambuf(void* (*alloc) (long), void(*free)(void*));
strstreambuf(char* sp, int len, char* tp);
strstreambuf(signed char* sp, int len, signed char* tp);
strstreambuf(unsigned char* sp, int len, unsigned char* tp);
```

The first version of the `strstreambuf` constructor takes no arguments and constructs an empty `strstreambuf` object in *dynamic mode*. Space will be allocated automatically to accommodate the characters that are put into the `strstreambuf` object. This space will be allocated using the operator `new` and deallocated using the operator `delete`. The characters that are already stored by the `strstreambuf` object may have to be copied when new space is allocated. If you know you are going to insert many characters into an `strstreambuf` object, you can give the I/O Stream Library an estimate of the size of the object before you create it by calling `strstreambuf::setbuf()`. See “setbuf” on page 94 for more details on `setbuf()`.

The second version of the strstreambuf constructor takes one argument and constructs an empty strstreambuf object in dynamic mode. The initial size of the stream buffer will be at least *bufsize* bytes.

The third version of the strstreambuf constructor takes two arguments and creates an empty strstreambuf object in dynamic mode. *alloc* is a pointer to the function that is used to allocate space. *alloc* is passed a long value that equals the number of bytes that it is supposed to allocate. If the value of *alloc* is 0, the operator new is used to allocate space. *free* is a pointer to the function that is used to free space. *free* is passed an argument that is a pointer to the array of bytes that *alloc* allocated. If *free* has a value of 0, the operator delete is used to free space.

The fourth, fifth, and sixth versions of the strstreambuf constructor take three arguments and construct a strstreambuf object with a stream buffer that begins at the position pointed to by *sp*. The nature of the stream buffer depends on the value of *len*:

- If *len* is positive, the *len* bytes following the position pointed to by *sp* make up the stream buffer.
- If *len* equals 0, *sp* points to the beginning of a null-terminated string, and the bytes of that string, excluding the terminating null character, will make up the stream buffer.
- If *len* is negative, the stream buffer has an indefinite length. The get pointer of the stream buffer is initialized to *sp*, and the put pointer is initialized to *tp*.

Regardless of the value of *len*, if the value of *tp* is 0, the get area will include the entire stream buffer, and insertions will cause errors.

Destructor for strstreambuf

~strstreambuf();

If freeze() has not been called for the strstreambuf object and a stream buffer is associated with the strstreambuf object, the strstreambuf destructor frees the space allocated by the strstreambuf constructor. The effect of the destructor depends on the constructor used to create the strstreambuf object:

- If you created the strstreambuf object using the constructor that takes two pointers to functions as arguments (see “Constructors for strstreambuf” on page 91 for more details), the destructor frees the space allocated by the destructor by calling the function pointed to by the second argument to the constructor.
- If you created the strstreambuf object using any of the other constructors, the destructor calls the delete operator to free the space allocated by the constructor.

doallocate

```
virtual int doallocate();
```

`doallocate()` attempts to allocate space for a stream buffer. If you created the `strstreambuf` object using the constructor that takes two pointers to functions as arguments (see “Constructors for strstreambuf” on page 91 for more details), `doallocate()` allocates space for the stream buffer by calling the function pointed to by the first argument to the constructor. Otherwise, `doallocate()` calls the operator `new` to allocate space for the stream buffer.

freeze

```
void freeze(int n=1);
```

`freeze()` controls whether the array that makes up a stream buffer can be deleted automatically. If *n* has a nonzero value, the array is not deleted automatically. If *n* equals 0, the array is deleted automatically when more space is needed or when the `strstreambuf` object is deleted. If you call `freeze()` with a nonzero argument for a `strstreambuf` object that was allocated in dynamic mode, any attempts to put characters in the stream buffer may result in errors. Therefore, you should avoid insertions to such stream buffers because the results are unpredictable. However, if you have a “frozen” stream buffer and you call `freeze()` with an argument equal to 0, you can put characters in the stream buffer again.

Only space that is acquired through dynamic allocation is ever freed.

overflow

```
virtual int overflow(int c);
```

`overflow()` causes the ultimate consumer to consume the characters in the put area and calls `setp()` to establish a new put area. The argument *c* is stored in the new put area if *c* is not equal to `EOF`.

str

```
char* str();
```

`str()` returns a pointer to the first character in the stream buffer and calls `freeze()` with a nonzero argument. Any attempts to put characters in the stream buffer may result in errors. If the `strstreambuf` object was created with an explicit array (that is, the `strstreambuf` constructor with three arguments was used), `str()` returns a pointer to that array. If the `strstreambuf` object was created in dynamic mode and nothing is stored in the array, `str()` may return 0.

seekoff

```
virtual streampos seekoff(
     streamoff so, ios::seek_dir dir, int mode);
```

`seekoff()` repositions the get or put pointer in the array of bytes in memory that serves as the ultimate producer or the ultimate consumer. For a text mode file, `seekoff()` returns the actual physical file position, because carriage return characters and end-of-file characters are discarded on input. Thus, there may not be a one-to-one correspondence between the characters in a stream and those in its external representation. For further details, see “Low-Level I/O” in the *IBM VisualAge C++ for OS/2 C Library Reference*, S25H-6964.

If you constructed the `strstreambuf` in dynamic mode (see “Constructors for strstreambuf” on page 91), the results of `seekoff()` are unpredictable. Therefore, do not use `seekoff()` with an `strstreambuf` object that you created in dynamic mode.

If you did not construct the `strstreambuf` object in dynamic mode, `seekoff()` attempts to reposition the get pointer or the put pointer, depending on the value of *mode*. If `ios::in` is set in *mode*, `seekoff()` repositions the get pointer. If `ios::out` is set in *mode*, `seekoff()` repositions the put pointer. If both `ios::in` and `ios::out` are set, `seekoff()` repositions both pointers.

`seekoff()` attempts to reposition the affected pointer to the value of *dir* + *so*. *dir* can have the following values:

- `ios::beg`: the beginning of the array in memory
- `ios::cur`: the current position in the array in memory
- `ios::end`: the end of the array in memory

If the value of *dir* + *so* is equal to or greater than the end of the array, the value is not valid and `seekoff()` returns `EOF`. Otherwise, `seekoff()` sets the affected pointer to this value and returns this value.

setbuf

```
virtual streambuf* setbuf(0, int bufsize);
```

`setbuf()` records *bufsize*. The next time that the `strstreambuf` object dynamically allocates a stream buffer, the stream buffer is at least *bufsize* bytes long.

Note: If you call `setbuf()` for an `strstreambuf` object, you must call it with the first argument equal to 0.

underflow

```
virtual int underflow();
```

If the get area is not empty, `underflow()` returns the first character in the get area. If the get area is empty, `underflow()` creates a new get area that is not empty and returns the first character. If no more characters are available in the ultimate producer, `underflow()` returns `EOF` and leaves the get area empty.

Part 3. Flat Collection Classes

This part contains detailed descriptions of the flat Collection Classes.

"Introduction to Flat Collections" on page 97 describes the common member functions for flat collections. Subsequent chapters describe individual collection classes.

For information on the organization of chapters that describe individual abstract data types, see "Format of Class Descriptions" on page 98.

Flat Collections

Introduction to Flat Collections

This chapter defines some of the terms used in describing the Collection Class Library classes and functions, describes the format of chapters that describe individual collections, and describes some types defined by the Collection Class Library.

Terms Used

CLASS_BASE_NAME	For constructor and destructor declarations, this term is used in place of the default implementation variant of a class. For example, the constructor `CLASS_BASE_NAME(...)` for a Bag, is really `IBag(...)`, because the default implementation variant of a bag is `IBag`.
CLASS_NAME	For member function declarations, this term is used in place of the class with template arguments. For example, if you want to use: `IBoolean operator != ( CLASS_NAME const& collection ) const;` for a Bag on BST Key Sorted Set, substitute `IBagOnBSTKeySortedSet<ElementName>` for `CLASS_NAME`.
equal element	Refers to equality of elements as defined by the equality operation or ordering relation provided for the element type (Chapter 9, “Element Functions and Key-Type Functions” in the *Open Class Library User's Guide* describes the purpose of the equality operation and ordering relation.) Where both equality operation and ordering relation are provided, the Collection Class Library may use either to determine element equality.
given ...	Refers to an argument of the described function, such as given element, given key, or given collection.
iteration order	The order in which elements are visited in `allElementsDo()` and `setToNext()` or `setToPrevious()`. In ordered collections, the element at position 1 will be visited first, then the element at position 2, and so on. Sorted collections, in particular, are visited following the ordering relation provided for the element type. In collections that are not ordered, the elements are visited in an arbitrary order. Each element is visited exactly once.

positioning property
The property of an element that is used to position the element in a collection. For key collections, the positioning property is key equality. For nonsequential collections with element equality, the positioning property is element equality. Other collections have no positioning property.

same key
Refers to equality of keys as defined by the equality operation or ordering relation provided for the key type. Where both equality operation and ordering relation are provided, the Collection Class Library may use either to determine key equality.

this collection
The collection to which a function is applied. Contrast with the *given* collection, which is an argument supplied to a function. *The collection* is synonymous with *this collection*.

undefined cursor
A cursor that may or may not be valid; there is no way to know whether the cursor is valid or not. An undefined cursor, even if it remains valid, may refer to a different element than before, or even to no element of the collection. Do not use cursors, once they become undefined, in functions that require the cursor to point to an element of the collection.

Format of Class Descriptions

Each chapter describing one or more Collection Classes consists of the following components:

- The chapter title, which usually refers to the kind of collection being discussed.
- A description of the collection's characteristics, such as whether the collection is sorted or unsorted, or whether the type and value of the elements are relevant.
- A textual example of using the collection in an application.
- Information on the class's derivation.
- A section on class implementation variants that provides some or all of the following information:
  - The default implementation, and the classes that you can use to alter the way the collection is implemented. These variant classes are based on other abstract data types within the Collection Class Library. For example, in the chapter on *heap* collections, the class `IHeapOnTabularSequence` is a heap collection based on `ITabularSequence`.
  - The names of the header files that correspond to particular implementation variants, so that you can include those files in your source code to make use of the implementation variants.

- A section on template arguments and required parameters that provides the following information:
  - Template arguments, which identify what parameters you must supply when you instantiate a particular implementation variant.
  - Required functions, which are functions that must be provided by the element type or key type you use for any implementation variant.
- A section on the reference class. The reference class allows you to make use of polymorphism. This section contains information on include files, template arguments and required functions similar to the information provided for the implementation variants described above. In general, reference classes do not put any additional requirements on the element type or key type. The requirements are those of the implementation variant used with the reference class.
- Optionally, a coding example to show you how to use the collection.

Required Functions

As described in Chapter 9, "Element Functions and Key-Type Functions" in the *Open Class Library User's Guide*, the Collection Classes require that you provide certain functions for the element type and key type. These functions are required by member functions of the Collection Class Library to manipulate elements and keys. The functions you must provide depend on the abstraction you use and on the implementation variant you choose. For example, you will usually need to provide a key access for all keyed abstractions, and for a hash table implementation you will need to provide a hash function.

Types Defined for the Collection Class Library

The following types are defined in `iglobals.h` or in header files included by `iglobals.h`:

```
typedef int IBoolean;

enum {
   false = 0,
   False = 0,
   true  = 1,
   True  = 1
};

typedef unsigned long INumber;
typedef unsigned long IPosition;

enum ITreeIterationOrder {IPreorder, IPostorder}; // for n-ary tree only

enum IExplicitInit { IINIT }; //for pointer classes only
```

Types Defined for the Collection Class Library

The `IExplicitInit` type is used by pointers from the pointer classes as a second constructor argument, in order to avoid using the constructor as an implicit conversion operator.

Note: If your environment defines another boolean type, use `IBoolean` wherever you want to refer to `Boolean` in the context of the Collection Class Library.

Flat Collection Member Functions

Each flat collection implements some or all of the member functions described in this chapter. Chapters on individual classes identify which functions are implemented for those classes.

Constructor

```
CLASS_BASE_NAME ( INumber numberOfElements = 100 ) ;
```

Constructs a collection. *numberOfElements* is the estimated maximum number of elements contained in the collection. The collection is unbounded and is initially empty. If the estimated maximum is exceeded, the collection is automatically enlarged.

Note: The collection constructor does not define whether any elements are constructed when the collection is constructed. For some classes, the element's default constructor may be invoked when the collection's constructor is invoked. This happens if a tabular or a diluted sequence implementation variant is used for a collection. The element's default constructor is used to allocate the required storage and initialize the elements. Therefore, a default constructor must be available for elements in such cases.

Exception: `IOutOfMemory`

Copy Constructor

```
CLASS_BASE_NAME ( CLASS_NAME const& collection ) ;
```

Constructs a collection and copies all elements from the given collection into the collection as described for "addAllFrom" on page 104.

Exception: `IOutOfMemory`

Destructor

```
~CLASS_BASE_NAME ( ) ;
```

Removes all elements from the collection. Destructors are called for all elements contained in the collection and for elements that have been constructed in advance.

Side Effects: All cursors of the collection become undefined.

Flat Collection Member Functions

operator!=

```
IBoolean operator!= ( CLASS_NAME const& collection ) const;
```

Returns `True` if the given collection is not equal to the collection. For a definition of equality for collections, see “operator==.”

operator=

```
CLASS_NAME& operator= ( CLASS_NAME const& collection ) ;
```

Copies the given collection to the collection. Removes all elements from the collection and adds the elements from the given collection as described for “addAllFrom” on page 104.

Preconditions

- If the collection is bounded, `numberOfElements()` of the given collection must be less than `maxNumberOfElements()` of this collection.

Side Effects

- All cursors of this collection become undefined.
- Collection classes supporting Visual Builder send a `modifiedId` notification.

Return Value: Returns a reference to the collection.

Exceptions

- `IOutOfMemory`
- `IFullException`, if the collection is bounded

operator==

```
IBoolean operator== ( CLASS_NAME const& collection ) const;
```

Returns `True` if the given collection is equal to the collection. Two collections are equal if the number of elements in each collection is the same, and if the condition for the collection is described in the following list:

Type of Collection	Condition
Unique Elements	If the collections have unique elements, any element that occurs in one collection must occur in the other collection.
Non-Unique Elements	If an element has *n* occurrences in one collection, it must have exactly *n* occurrences in the other collection.
Sequential	The ordering of the elements is the same for both collections.

add

```
IBoolean add ( Element const& element ) ;

IBoolean add ( Element const& element,
   ICursor& cursor ) ;
```

If the collection is unique (with respect to elements or keys) and the element or key is already contained in the collection, sets the cursor to the existing element in the collection without adding the element. Otherwise, it adds the element to the collection and sets the cursor to the added element. In sequential collections, the given element is added as the last element. In sorted collections, the element is added at a position determined by the element or key value. Adding an element will either use the element's copy constructor or the assignment operator provided for the element type, depending on the implementation variant you choose. See “contains” on page 113 for the definition of element or key containment.

Preconditions

- The cursor must belong to the collection.
- If the collection is bounded and unique, the element or key must exist or (`numberOfElements() < maxNumberOfElements()`).
- If the collection is bounded and nonunique, (`numberOfElements() < maxNumberOfElements()`).
- If the collection is a map or a sorted map and contains an element with the same key as the given element, this element must be equal to the given element.

Side Effects

- If an element was added, all cursors of this collection, except the given cursor, become undefined.
- If an element was added, collection classes supporting Visual Builder send an `addedId` notification.

Return Value: Returns `True` if the element was added.

Exceptions

- `IOutOfMemory`
- `ICursorInvalidException`
- `IFullException`, if the collection is bounded
- `IKeyAlreadyExistsException`, if the collection is a map or a sorted map

Flat Collection Member Functions

addAllFrom

```
void addAllFrom ( CLASS_NAME const& collection ) ;

void addAllFrom (
   IACollection <Element> const& collection ) ;
```

Adds (copies) all elements of the given collection to the collection. The elements are added in the iteration order of the given collection. The contents of the elements, not the pointers to the elements, are copied. The elements are added according to the definition of add for this collection. The given collection is not changed.

Preconditions: Because the elements are added one by one, the following preconditions are tested for each individual add operation:

- If the collection is bounded and unique, the element or key must exist or (`numberOfElements() < maxNumberOfElements()`).
- If the collection is bounded and nonunique, (`numberOfElements() < maxNumberOfElements()`).
- If the collection is a map or a sorted map and contains an element with the same key as the given element, this element must be equal to the given element.

Side Effects

- If any elements were added, all cursors of this collection become undefined.
- If any elements were added, collection classes supporting Visual Builder send a `modifiedId` notification.

Exceptions

- `IOutOfMemory`
- `IIdenticalCollectionException`
- `IFullException`, if the collection is bounded
- `IKeyAlreadyExistsException`, if the collection is a map or a sorted map

addAsFirst

```
void addAsFirst ( Element const& element ) ;

void addAsFirst ( Element const& element,
   ICursor& cursor ) ;
```

Adds the element to the collection as the first element in sequential order. Sets the cursor to the added element.

Preconditions

- The cursor must belong to the collection.
- If the collection is bounded, (`numberOfElements() < maxNumberOfElements()`).

Side Effects

- All cursors of this collection, except the given cursor, become undefined.
- If an element was added, collection classes supporting Visual Builder send an `addedId` notification.

Exceptions

- `ICursorInvalidException`
- `IOutOfMemory`
- `IFullException`, if the collection is bounded

addAsLast

```
void addAsLast ( Element const& element ) ;

void addAsLast ( Element const& element,
   ICursor& cursor ) ;
```

Adds the element to the collection as the last element in sequential order. Sets the cursor to the added element.

Preconditions

- The cursor must belong to the collection.
- If the collection is bounded, (`numberOfElements() < maxNumberOfElements()`).

Side Effects

- All cursors of this collection, except the given cursor, become undefined.
- If an element was added, collection classes supporting Visual Builder send an `addedId` notification.

All cursors of this collection, except the given cursor, become undefined.

Exceptions

- `ICursorInvalidException`
- `IOutOfMemory`
- `IFullException`, if the collection is bounded

addAsNext

```
void addAsNext ( Element const& element,
   ICursor& cursor ) ;
```

Adds the element to the collection as the element following element pointed to by the cursor. Sets the cursor to the added element.

Preconditions

- The cursor must belong to the collection and must point to an element of the collection.
- If the collection is bounded, (`numberOfElements() < maxNumberOfElements()`).

Side Effects

- All cursors of this collection, except the given cursor, become undefined.
- If an element was added, collection classes supporting Visual Builder send an `addedId` notification.

Exceptions

- `IOutOfMemory`
- `ICursorInvalidException`
- `IFullException`, if the collection is bounded

addAsPrevious

```
void addAsPrevious ( Element const& element,
   ICursor& cursor ) ;
```

Adds the element to the collection as the element preceding the element pointed to by the cursor. Sets the cursor to the added element.

Preconditions

- The cursor must belong to the collection and must point to an element of the collection.
- If the collection is bounded, (`numberOfElements() < maxNumberOfElements()`).

Side Effects

- All cursors of this collection, except the given cursor, become undefined.
- If an element was added, collection classes supporting Visual Builder send an `addedId` notification.

Exceptions

- `IOutOfMemory`
- `ICursorInvalidException`
- `IFullException`, if the collection is bounded

addAtPosition

```
void addAtPosition ( IPosition position,
   Element const& element ) ;

void addAtPosition ( IPosition position,
   Element const& element, ICursor& cursor ) ;
```

Adds the element at the given position to the collection, and sets the cursor to the added element. If an element exists at the given position, the new element is added as the element preceding the existing element.

Preconditions

- The cursor must belong to the collection.
- (1 ≤ *position* ≤ `numberOfElements + 1`).
- If the collection is bounded, (`numberOfElements() < maxNumberOfElements()`).

Side Effects

- All cursors of this collection, except the given cursor, become undefined.
- If an element was added, collection classes supporting Visual Builder send an `addedId` notification.

Exceptions

- `IOutOfMemory`
- `ICursorInvalidException`
- `IPositionInvalidException`
- `IFullException`, if the collection is bounded

addDifference

```
void addDifference ( CLASS_NAME const& collection1,
   CLASS_NAME const& collection2 ) ;
```

Creates the difference between the two given collections, and adds this difference to the collection. The contents of the added elements, not the pointers to those elements, are copied.

For a definition of the difference between two collections, see “differenceWith” on page 114.

Preconditions: Because the elements are added one by one, the following preconditions are tested for each individual addition.

- If the collection is bounded and unique, the element or key must exist or (`numberOfElements() < maxNumberOfElements()`).
- If the collection is bounded and nonunique, (`numberOfElements() < maxNumberOfElements()`).
- If the collection is a map or a sorted map and contains an element with the same key as the given element, this element must be equal to the given element.

Side Effects

- If any elements were added, all cursors of this collection become undefined.
- If any elements were added, collection classes supporting Visual Builder send a `modifiedId` notification.

Exceptions

- `IOutOfMemory`
- `IFullException`, if the collection is bounded
- `IKeyAlreadyExistsException`, if the collection is a map or a sorted map

addIntersection

```
void addIntersection ( CLASS_NAME const& collection1,
   CLASS_NAME const& collection2 ) ;
```

Creates the intersection of the two given collections, and adds this intersection to the collection. The contents of the added elements, not the pointers to those elements, are copied.

For a definition of the intersection of two collections, see “intersectionWith” on page 117.

Preconditions: Because the elements are added one by one, the following preconditions are tested for each individual addition.

- If the collection is bounded and unique, the element or key must exist or (`numberOfElements() < maxNumberOfElements()`).
- If the collection is bounded and nonunique, (`numberOfElements() < maxNumberOfElements()`).
- If the collection is a map or a sorted map and contains an element with the same key as the given element, this element must be equal to the given element.

Side Effects

- If any elements were added, all cursors of this collection become undefined.
- If any elements were added, collection classes supporting Visual Builder send a `modifiedId` notification.

Exceptions

- `IOutOfMemory`
- `IFullException`, if the collection is bounded
- `IKeyAlreadyExistsException`, if the collection is a map or a sorted map

addOrReplaceElementWithKey

```
IBoolean addOrReplaceElementWithKey (
   Element const& element );

IBoolean addOrReplaceElementWithKey (
   Element const& element, ICursor& cursor ) ;
```

If an element is contained in the collection where the key is equal to the key of the given element, sets the cursor to this element in the collection and replaces it with the given element. Otherwise, it adds the given element to the collection, and sets the cursor to the added element. If the given element is added, the contents of the element, not a pointer to it, is added.

Preconditions

- The cursor must belong to the collection.
- If the collection is bounded, an element with the given key must be contained in the collection, or (`numberOfElements() < maxNumberOfElements()`).

Side Effects

- If the element was added, all cursors of this collection, except the given cursor, become undefined.
- If the element was added, collection classes supporting Visual Builder send a `replacedId` notification.

Return Value: Returns `True` if the element was added. Returns `False` if the element was replaced.

Exceptions

- `IOutOfMemory`
- `ICursorInvalidException`
- `IFullException`, if the collection is bounded

Flat Collection Member Functions

addUnion

```
void addUnion (  CLASS_NAME const& collection1,
   CLASS_NAME const& collection2 ) ;
```

Creates the union of the two given collections, and adds this union to the collection. The contents of the added elements, not the pointers to those elements, are copied.

For a definition of the union of two collections, see "unionWith" on page 132.

Preconditions: Because the elements are added one by one, the following preconditions are tested for each individual addition.

- If the collection is bounded and unique, the element or key must exist or (`numberOfElements() < maxNumberOfElements()`).
- If the collection is bounded and nonunique, (`numberOfElements() < maxNumberOfElements()`).
- If the collection is a map or a sorted map and contains an element with the same key as the given element, this element must be equal to the given element.

Side Effects

- If any elements were added, all cursors of this collection become undefined.
- If any elements were added, collection classes supporting Visual Builder send a `modifiedId` notification.

Exceptions

- `IOutOfMemory`
- `IFullException`, if the collection is bounded
- `IKeyAlreadyExistsException`, if the collection is a map or a sorted map

allElementsDo

```
IBoolean allElementsDo (
   IBoolean (*function) (Element&, void*),
   void* additionalArgument = 0 ) ;

IBoolean allElementsDo (
   IBoolean (*function) (Element const&, void*),
   void* additionalArgument = 0 ) const;
```

Calls the given function for all elements in the collection until the given function returns `False`. The elements are visited in iteration order. Additional arguments can be passed to the given function using `additionalArgument`. The additional argument defaults to zero if no additional argument is given.

Notes:

1. The given function must not remove elements from or add them to the collection. If you want to remove elements, you can use the `removeAll()` function with a property argument. For further information see "removeAll" on page 125.
2. For the non-**const** version of `allElementsDo()`, the given function must not manipulate the element in the collection in a way that changes the positioning property of the element.

Return Value: Returns `True` if the given function returns `True` for every element it is applied to.

allElementsDo

```
IBoolean allElementsDo (
   IIterator <Element>& iterator ) ;

IBoolean allElementsDo (
   IConstantIterator <Element>& iterator ) const;
```

Calls the `applyTo()` function of the given iterator for all elements of the collection until the `applyTo()` function returns `False`. The elements are visited in iteration order. Additional arguments may be passed as arguments to the constructor of the derived iterator class. (For further details, see "Iteration Using Iterators" in the *Open Class Library User's Guide*.)

Notes:

1. The `applyTo()` function must not remove elements from or add elements to the collection. If you want to remove elements, you can use the `removeAll()` function with a property argument. For further information, see "removeAll" on page 125.
2. For the non-**const** version of `allElementsDo()`, the `applyTo()` function must not manipulate the element in the collection in a way that changes the positioning property of the element.

Return Value: Returns `True` if the `applyTo()` function returns `True` for every element it is applied to.

Flat Collection Member Functions

anyElement

```
Element const& anyElement ( ) const;
```

Returns a reference to an arbitrary element of the collection.

Precondition: The collection must not be empty.

Exception: `IEmptyException`

compare

```
long compare ( CLASS_NAME const& collection,
   long (*comparisonFunction)
 (Element const& element1,Element const& element2)
   ) const;
```

Compares the collection with the given collection. Comparison yields <0 if the collection is less than the given collection, 0 if the collection is equal to the given collection, and >0 if the collection is greater than the given collection. Comparison is defined by the first pair of corresponding elements, in both collections, that are not equal. If such a pair exists, the collection with the greater element is the greater one. Otherwise, the collection with more elements is the greater one.

Notes:

1. The given comparison function must return a result according to the following rules:

>0	if (element1 > element2)
0	if (element1 == element2)
<0	if (element1 < element2)

2. For elements of type `char*`, `compare()` is not locale-sensitive by default. Because it uses `strcmp()` and not `strcoll()`, it compares the binary values representing the characters, and is not based on the `LC_COLLATE` category of the current locale. Its results are reliable only for code pages and character sets in which the collating sequence matches the sequence of binary representations.

Return Value: Returns the result of the collection comparison.

contains

```
IBoolean contains ( Element const& element ) const;
```

Returns True if the collection contains an element equal to the given element.

containsAllFrom

```
IBoolean containsAllFrom (
   CLASS_NAME const& collection ) const;

IBoolean containsAllFrom (
   IACollection <Element> const& collection ) const;
```

Returns True if all the elements of the given collection are contained in the collection. The definition of containment is described in “contains.”

containsAllKeysFrom

```
IBoolean containsAllKeysFrom (
   CLASS_NAME const& collection ) const;

IBoolean containsAllKeysFrom (
   IACollection <Element> const& collection ) const;
```

Returns True if all of the keys of the given collection are contained in the collection.

containsElementWithKey

```
IBoolean containsElementWithKey ( Key const& key ) const;
```

Returns True if the collection contains an element with the same key as the given key.

copy

```
void copy ( IACollection <Element> const& collection ) ;
```

Copies the given collection to this collection. copy() removes all elements from this collection, and adds the elements from the given collection. For information on how adding is done, see “addAllFrom” on page 104.

Note: The given collection may be of a concrete type other than the collection itself. In this case, copying implicitly performs a conversion. If, for example, the given collection is a bag and the collection itself is a set, elements with multiple occurrences in the copied bag will only occur once in the resulting set.

Preconditions: Because the elements are copied one by one, the following preconditions are tested for each individual copy operation:

- If the collection is bounded and unique, the element or key must exist or (numberOfElements() < maxNumberOfElements()).
- If the collection is bounded and nonunique, (numberOfElements() < maxNumberOfElements()).

- If the collection is a map or a sorted map and contains an element with the same key as the given element, this element must be equal to the given element.

Side Effects

- All cursors of this collection become undefined.
- If any elements were copied, collection classes supporting Visual Builder send a `modifiedId` notification.

Exceptions

- `IOutOfMemory`
- `IFullException`, if the collection is bounded
- `IKeyAlreadyExistsException`, if the collection has unique keys. This exception may be thrown, for example, when copying a bag into a map.

dequeue

```
void dequeue ( ) ;

void dequeue ( Element& element ) ;
```

Copies the first element of the collection to the given element, and removes it from the collection.

Precondition: The collection must not be empty.

Side Effects

- All cursors of this collection become undefined.
- If the element is removed, collection classes supporting Visual Builder send a `modifiedId` notification.

Exception: `IEmptyException`

differenceWith

```
void differenceWith ( CLASS_NAME const& collection ) ;
```

Makes the collection the difference between the collection and the given collection. The *difference* of A and B (A minus B) is the set of elements that are contained in A but not in B.

The following rule applies for bags with duplicate elements: If bag P contains the element X *m* times and bag Q contains the element X *n* times, the *difference* of P and Q contains the element X *m-n* times if *m > n*, and zero times if *m≤n*.

Side Effects

- If any elements were removed, all cursors of this collection become undefined.
- If the element is removed, collection classes supporting Visual Builder send a `modifiedId` notification.

elementAt

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

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

Returns a reference to the element pointed to by the given cursor.

Note: For the version of `elementAt()` *without* the **`const`** suffix, do not manipulate the element or the key of the element in the collection in a way that changes the positioning property of the element.

Precondition: The cursor must belong to the collection and must point to an element of the collection.

Exception: `ICursorInvalidException`

elementAtPosition

```
Element const& elementAtPosition (
   IPosition position ) const;
```

Returns a reference to the element at the given position in the collection.

Position 1 specifies the first element.

`Position` must be a valid position in the collection; that is, (1 ≤ *`position`* ≤ `numberOfElements()`).

Precondition: (1 ≤ *`position`* ≤ `numberOfElements()`).

Exception: `IPositionInvalidException`

elementWithKey

```
Element& elementWithKey ( Key const& key ) ;
```

```
Element const& elementWithKey ( Key const& key ) const;
```

Returns a reference to an element specified by the key.

Notes:

1. For the version of `elementWithKey()` *without* a **`const`** suffix, do not manipulate the element in the collection in a way that changes the positioning property of the element.
2. If there are several elements with the given key, an arbitrary one is returned.

Precondition: The given key is contained in the collection.

Exception: `INotContainsKeyException`

enqueue

```
void enqueue ( Element const& element ) ;

void enqueue ( Element const& element,
   ICursor& cursor ) ;
```

Adds the element to the collection, and sets the cursor to the added element. For ordinary queues, the given element is added as the last element. For priority queues, the element is added at a position determined by the ordering relation provided for the element or key type.

Preconditions

- The cursor must belong to the collection.
- If the collection is bounded, (`numberOfElements() < maxNumberOfElements()`).

Side Effects

- All cursors of this collection except the given cursor become undefined.
- If the element is added, collection classes supporting Visual Builder send a `modifiedId` notification.

Exceptions

- `IOutOfMemory`
- `ICursorInvalidException`
- `IFullException`, if the collection is bounded

firstElement `Element const& firstElement ( ) const;`

Returns a reference to the first element of the collection.

Precondition: The collection must not be empty.

Exception: `IEmptyException`

intersectionWith

`void intersectionWith ( CLASS_NAME const& collection ) ;`

Makes the collection the intersection of the collection and the given collection. The *intersection* of A and B is the set of elements that is contained in both A and B.

The following rule applies for bags with duplicate elements: If bag P contains the element X *m* times and bag Q contains the element X *n* times, the *intersection* of P and Q contains the element X `MIN(m,n)` times.

Side Effects

- If any elements were removed, all cursors of this collection become undefined.
- If any elements were removed, collection classes supporting Visual Builder send a `modifiedId` notification.

isBounded `IBoolean isBounded ( ) const;`

Returns `True` if the collection is bounded.

isEmpty `IBoolean isEmpty ( ) const;`

Returns `True` if the collection is empty.

isFirst `IBoolean isFirst ( ICursor const& cursor ) const;`

Returns `True` if the given cursor points to the first element of the collection.

Preconditions: The cursor must belong to the collection and must point to an element of the collection.

Exception: `ICursorInvalidException`

Flat Collection Member Functions

isFull

```
IBoolean isFull ( ) const;
```

Returns `True` if the collection is bounded and contains the maximum number of elements; that is, if `(numberOfElements() == maxNumberOfElements())`.

isLast

```
IBoolean isLast ( ICursor const& cursor ) const;
```

Returns `True` if the given cursor points to the last element of the collection.

Preconditions: The cursor must belong to the collection and must point to an element of the collection.

Exception: `ICursorInvalidException`

key

```
Key const& key ( Element const& element ) const;
```

Returns a reference to the key of the given element using the `key()` function provided for the element type.

lastElement

```
Element const& lastElement ( ) const;
```

Returns a reference to the last element of the collection.

Precondition: The collection must not be empty.

Exception: `IEmptyException`

locate

```
IBoolean locate ( Element const& element,
   ICursor& cursor ) const;
```

Locates an element in the collection that is equal to the given element. Sets the cursor to point to the element in the collection, or invalidates the cursor if no such element exists.

If the collection contains several such elements, the first element in iteration order is located.

Precondition: The cursor must belong to the collection.

Return Value: Returns `True` if an element was found.

Exceptions: `ICursorInvalidException`

locateElementWithKey

```
IBoolean locateElementWithKey ( Key const& key,
   ICursor& cursor ) const;
```

Locates an element in the collection with the same key as the given key. Sets the cursor to point to the element in the collection, or invalidates the cursor if no such element exists.

If the collection contains several such elements, the first element in iteration order is located.

Precondition: The cursor must belong to the collection.

Return Value: Returns `True` if an element was found.

Exception: `ICursorInvalidException`

locateFirst

```
IBoolean locateFirst ( Element const& element,
   ICursor& cursor ) const;
```

Locates the first element in iteration order in the collection that is equal to the given element. Sets the cursor to the located element, or invalidates the cursor if no such element exists.

Precondition: The cursor must belong to the collection.

Return Value: Returns `True` if an element was found.

Exception: `ICursorInvalidException`

locateLast

```
IBoolean locateLast ( Element const& element,
   ICursor& cursor ) const;
```

Locates the last element in iteration order in the collection that is equal to the given element. Sets the cursor to the located element, or invalidates the cursor if no such element exists.

Precondition: The cursor must belong to the collection.

Return Value: Returns `True` if an element was found.

Exception: `ICursorInvalidException`

Flat Collection Member Functions

locateNext

```
IBoolean locateNext ( Element const& element,
   ICursor& cursor ) const;
```

Locates the next element in iteration order in the collection that is equal to the given element, starting at the element next to the one pointed to by the given cursor. Sets the cursor to point to the element in the collection. The cursor is invalidated if the end of the collection is reached and no more occurrences of the given element are left to be visited.

Note: If you code a call to `locateFirst()` and a set of calls to `locateNext()`, each occurrence of an element will be visited exactly once in iteration order.

Precondition: The cursor must belong to the collection and must point to an element of the collection.

Return Value: Returns `True` if an element was found.

Exception: `ICursorInvalidException`

locateNextElementWithKey

```
IBoolean locateNextElementWithKey (
   Key const& key, ICursor& cursor ) const;
```

Locates the next element in iteration order in the collection with the given key, starting at the element next to the one pointed to by the given cursor. Sets the cursor to point to the element in the collection. The cursor is invalidated if the end of the collection is reached and no more occurrences of such an element are left to be visited.

Note: If you code a call to `locateFirst()` and a set of calls to `locateNextElementWithKey()`, each occurrence of an element will be visited exactly once in iteration order.

Preconditions: The cursor must belong to the collection and must point to an element of the collection.

Return Value: Returns `True` if an element was found.

Exception: `ICursorInvalidException`

locateOrAdd

```
IBoolean locateOrAdd ( Element const& element ) ;

IBoolean locateOrAdd ( Element const& element,
   ICursor& cursor ) ;
```

Locates an element in the collection that is equal to the given element. (See "locate" on page 118 for details on `locate()`.) If no such element is found, `locateOrAdd()` adds the element as described in "add" on page 103. The cursor is set to the located or added element.

Note: This method may be more efficient than using `locate()` followed by a conditionally called `add()`.

Preconditions

- The cursor must belong to the collection.
- If the collection is a map or a sorted map and contains an element with the same key as the given element, this element must be equal to the given element.
- The element or key must exist, or (`numberOfElements() < maxNumberOfElements()`).

Side Effects

- If the element was added, all cursors of this collection, except the given cursor, become undefined.
- If the element was added, collection classes supporting Visual Builder send an `addedId` notification.

Return Value: Returns `True` if the element was located. Returns `False` if the element could not be located but had to be added.

Exceptions

- `IOutOfMemory`
- `ICursorInvalidException`
- `IFullException`, if the collection is bounded
- `IKeyAlreadyExistsException`, if the collection is a map or a sorted map

Flat Collection Member Functions

locateOrAddElementWithKey

```
IBoolean locateOrAddElementWithKey (
   Element const& element ) ;

IBoolean locateOrAddElementWithKey (
   Element const& element; ICursor& cursor ) ;
```

Locates an element in the collection with the given key as described for the `locateElementWithKey()` function. If no such element exists, `locateOrAddElementWithKey()` adds the element as described in "add" on page 103. The cursor is set to the located or added element.

Preconditions

- If the collection is bounded and an element with the given key is not already contained, (`numberOfElements() < maxNumberOfElements()`).
- The cursor must belong to the collection.

Side Effects

- If the element was added, all cursors of this collection, except the given cursor, become undefined.
- If the element was added, collection classes supporting Visual Builder send an `addedId` notification.

Return Value: Returns `True` if the element was located. Returns `False` if the element could not be located but had to be added.

Exceptions

- `IOutOfMemory`
- `ICursorInvalidException`
- `IFullException`, if the collection is bounded

locatePrevious

```
IBoolean locatePrevious ( Element const& element,
   ICursor& cursor ) const;
```

Locates the previous element in iteration order that is equal to the given element, beginning at the element previous to the one specified by the given cursor and moving in reverse iteration order through the elements. Sets the cursor to the located element, or invalidates the cursor if no such element exists.

Preconditions: The cursor must belong to the collection and must point to an element of the collection.

Return Value: Returns `True` if an element was found.

Exceptions: `ICursorInvalidException`

maxNumberOfElements

```
INumber maxNumberOfElements ( ) const;
```

Returns the maximum number of elements the collection can contain.

Precondition: The collection is bounded.

Exceptions: `INotBoundedException`

newCursor

```
Cursor* newCursor ( ) const;
```

Creates a cursor for the collection and returns a pointer to the cursor. The cursor is initially not valid.

Exception: `IOutOfMemory`

numberOfDifferentElements

```
INumber numberOfDifferentElements ( ) const;
```

Returns the number of different elements in the collection.

numberOfDifferentKeys

```
INumber numberOfDifferentKeys ( ) const;
```

Returns the number of different keys in the collection.

numberOfElements

```
INumber numberOfElements ( ) const;
```

Returns the number of elements the collection contains.

numberOfElementsWithKey

```
INumber numberOfElementsWithKey (
   Key const& key ) const;
```

Returns the number of elements in the collection with the given key.

Flat Collection Member Functions

numberOfOccurrences

```
INumber numberOfOccurrences (
   Element const& element ) const;
```

Returns the number of occurrences of the given element in the collection.

pop

```
void pop ( ) ;
```

```
void pop ( Element& element ) ;
```

Copies the last element of the collection to the given element, and removes it from the collection.

Precondition: The collection must not be empty.

Side Effects

- All cursors of this collection become undefined.
- If the element was removed from the collection, collection classes supporting Visual Builder send a `removedId` notification.

Exception: `IEmptyException`

push

```
void push ( Element const& element ) ;
```

```
void push ( Element const& element,
   ICursor& cursor ) ;
```

Adds the element to the collection as the last element (as defined for "add" on page 103), and sets the cursor to the added element.

Preconditions

- The cursor must belong to the collection.
- If the collection is bounded, (`numberOfElements() < maxNumberOfElements()`).

Side Effects

- All cursors of this collection, except the given cursor, become undefined.
- If the element was added to the collection, collection classes supporting Visual Builder send an `addedId` notification.

Exceptions

- `IOutOfMemory`
- `ICursorInvalidException`
- `IFullException`, if the collection is bounded

remove

```
IBoolean remove ( Element const& element ) ;
```

Removes an element in the collection that is equal to the given element. If no such element exists, the collection remains unchanged. In collections with nonunique elements, an arbitrary occurrence of the given element will be removed. Element destructors are called as described in “removeAt” on page 126.

Side Effects

- If an element was removed, all cursors of this collection become undefined.
- If an element was removed, collection classes supporting Visual Builder send a `removedId` notification.

Return Value: Returns `True` if an element was removed.

removeAll

```
void removeAll ( ) ;
```

Removes all elements from the collection. Element destructors are called as described in “removeAt” on page 126.

Side Effects

- All cursors of this collection become undefined.
- Collection classes supporting Visual Builder send a `modifiedId` notification.

removeAll

```
INumber removeAll (
   IBoolean (*property) (Element const&, void*),
   void* additionalArgument = 0 ) ;
```

Removes all elements from this collection for which the given property function returns `True`. Additional arguments can be passed to the given property function using `additionalArgument`. The additional argument defaults to zero if no additional argument is given. Element destructors are called as described in “removeAt” on page 126.

Side Effects

- If any elements were removed, all cursors of this collection become undefined.
- If any elements were removed, collection classes supporting Visual Builder send a `modifiedId` notification.

Return Value: The number of elements removed.

Flat Collection Member Functions

removeAllElementsWithKey

```
INumber removeAllElementsWithKey (
   Key const& key ) ;
```

Removes all elements from the collection with the same key as the given key. Element destructors are called as described in "removeAt."

Side Effects

- If any elements were removed, all cursors of this collection become undefined.
- If any elements were removed, collection classes supporting Visual Builder send a `removedId` notification.

Return Value: The number of elements removed.

removeAllOccurrences

```
INumber removeAllOccurrences ( Element const& element ) ;
```

Removes all elements from the collection that are equal to the given element, and returns the number of elements removed. Element destructors are called as described in "removeAt."

Side Effects

- If any elements were removed, all cursors of this collection become undefined.
- If any elements were removed, collection classes supporting Visual Builder send a `modifiedId` notification.

removeAt

```
void removeAt ( ICursor& cursor ) ;
```

Removes the element pointed to by the given cursor. The given cursor is invalidated.

Note: It is undefined whether the destructor for the removed element is called or whether the element will only be destructed with the collection destructor. For example, in a tabular implementation, a destructor will only be called when the whole collection is destructed, not when a single element is removed.

Preconditions: The cursor must belong to the collection and must point to an element of the collection.

Side Effects

- All cursors of this collection, except the given cursor, become undefined.
- If an element was removed, collection classes supporting Visual Builder send a `removedId` notification.

Exception: `ICursorInvalidException`

removeAtPosition

```
void removeAtPosition ( IPosition position ) ;
```

Removes the element from the collection that is at the given position. Element destructors are called as described in “removeAt” on page 126.

The first element of the collection has position 1.

Precondition: Position must be a valid position in the collection; that is, (1 ≤ *position* ≤ `numberOfElements()`).

Side Effects

- All cursors of this collection become undefined.
- Collection classes supporting Visual Builder send a `removedId` notification.

Exception: `IPositionInvalidException`

removeElementWithKey

```
IBoolean removeElementWithKey ( Key const& key ) ;
```

Removes an element from the collection with the same key as the given key. If no such element exists, the collection remains unchanged. In collections with nonunique elements, an arbitrary occurrence of such an element will be removed. Element destructors are called as described in “removeAt” on page 126.

Side Effects

- If an element was removed, all cursors of this collection become undefined.
- If an element was removed, collection classes supporting Visual Builder send a `removedId` notification.

Return Value: Returns `True` if an element was removed.

removeFirst

```
void removeFirst ( ) ;
```

Removes the first element from the collection. Element destructors are called as described in “removeAt” on page 126.

Precondition: The collection must not be empty.

Side Effects

- All cursors of this collection become undefined.
- If an element was removed, collection classes supporting Visual Builder send a `removedId` notification.

Exception: `IEmptyException`

removeLast

```
void removeLast ( ) ;
```

Removes the last element from the collection. Element destructors are called as described in "removeAt" on page 126.

Precondition: The collection must not be empty.

Side Effects

- All cursors of this collection become undefined.
- If an element was removed, collection classes supporting Visual Builder send a `removedId` notification.

Exception: `IEmptyException`

replaceAt

```
void replaceAt ( ICursor const& cursor,
   Element const& element ) ;
```

Replaces the element pointed to by the cursor with the given element.

Preconditions

- The cursor must belong to the collection and must point to an element of the collection.
- The given element must have the same positioning property as the replaced element.

Side Effect: Collection classes supporting Visual Builder send a `replacedId` notification.

Exceptions

- `ICursorInvalidException`
- `IInvalidReplacementException`

replaceElementWithKey

```
IBoolean replaceElementWithKey ( Element const& element ) ;

IBoolean replaceElementWithKey ( Element const& element,
   ICursor& cursor ) ;
```

Replaces an element with the same key as the given element by the given element, and sets the cursor to this element. If no such element exists, it invalidates the cursor. In collections with nonunique elements, an arbitrary occurrence of such an element will be replaced.

Precondition: The cursor must belong to the collection.

Side Effect: Collection classes supporting Visual Builder send a `replacedId` notification.

Return Value: Returns `True` if an element was replaced.

Exceptions: `ICursorInvalidException`

setToFirst

```
IBoolean setToFirst ( ICursor& cursor ) const;
```

Sets the cursor to the first element of the collection in iteration order. If the collection is empty (if no first element exists), it invalidates the given cursor.

Precondition: The cursor must belong to the collection.

Return Value: Returns `True` if the collection is not empty.

Exception: `ICursorInvalidException`

setToLast

```
IBoolean setToLast ( ICursor& cursor ) const;
```

Sets the cursor to the last element of the collection in iteration order. If the collection is empty (if no last element exists), the given cursor is no longer valid.

Precondition: The cursor must belong to the collection.

Return Value: Returns `True` if the collection is not empty.

Exceptions: `ICursorInvalidException`

Flat Collection Member Functions

setToNext

```
IBoolean setToNext ( ICursor& cursor ) const;
```

Sets the cursor to the next element in the collection in iteration order. If no more elements are left to be visited, the given cursor will no longer be valid.

Precondition: The cursor must belong to the collection and must point to an element.

Return Value: Returns `True` if there is a next element.

Exceptions: `ICursorInvalidException`

setToNextDifferentElement

```
IBoolean setToNextDifferentElement (
   ICursor& cursor ) const;
```

Sets the cursor to the next element in iteration order in the collection that is different from the element pointed to by the given cursor. If no more elements are left to be visited, the given cursor will no longer be valid.

Precondition: The cursor must belong to the collection and must point to an element of the collection.

Return Value: Returns `True` if a subsequent element was found that is different.

Exception: `ICursorInvalidException`

setToNextWithDifferentKey

```
IBoolean setToNextWithDifferentKey ( ICursor& cursor ) const;
```

Sets the cursor to the next element in the collection in iteration order with a key different from the key of the element pointed to by the given cursor. If no such element exists, the given cursor is no longer valid.

Preconditions: The cursor must belong to the collection and must point to an element of the collection.

Return Value: Returns `True` if a subsequent element was found whose key is different from the current key.

Exception: `ICursorInvalidException`

setToPosition

```
void setToPosition ( IPosition position,
   ICursor& cursor ) const;
```

Sets the cursor to the element at the given position. Position 1 specifies the first element.

Precondition

- The cursor must belong to the collection.
- Position must be a valid position in the collection; that is, `(1 ≤ position ≤ numberOfElements())`.

Exceptions

- `ICursorInvalidException`
- `IPositionInvalidException`

setToPrevious

```
IBoolean setToPrevious ( ICursor& cursor ) const;
```

Sets the cursor to the previous element in iteration order, or invalidates the cursor if no such element exists.

Preconditions: The cursor must belong to the collection and must point to an element of the collection.

Return Value: Returns `True` if a previous element exists.

Exception: `ICursorInvalidException`

sort

```
void sort ( long (*comparisonFunction)
   (Element const& element1, Element const& element2) );
```

Sorts the collection so that the elements occur in ascending order. The relation of two elements is defined by the *comparisonFunction*, which you provide.

Note: The *comparisonFunction* must deliver a result according to the following rules:

>0	if `(element1 > element2)`
0	if `(element1 == element2)`
<0	if `(element1 < element2)`

Side Effects

- All cursors of this collection become undefined.
- Collection classes supporting Visual Builder send a `modifiedId` notification.

Flat Collection Member Functions

top

```
Element const& top ( ) const;
```

Returns a reference to the last element of the collection.

Precondition: The collection must not be empty.

Exception: `IEmptyException`

unionWith

```
void unionWith (
   CLASS_NAME const& collection ) ;
```

Makes the collection the union of the collection and the given collection. The *union* of A and B is the set of elements that are members of A or B or both.

The following rule applies for bags with duplicate elements: If bag P contains the element X *m* times and bag Q contains the element X *n* times, the *union* of P and Q contains the element X *m*+*n* times.

Preconditions: Because the elements from the given collection are added to the collection one by one, the following preconditions are tested for each individual add operation :

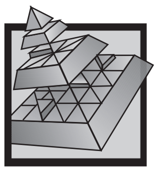
- If the collection is bounded and unique, the element or key must exist or (`numberOfElements() < maxNumberOfElements()`).
- If the collection is bounded and nonunique, (`numberOfElements() < maxNumberOfElements()`).
- If the collection is a map or a sorted map and contains an element with the same key as the given element, this element must be equal to the given element.

Side Effects

- If any elements were added to the collection, all cursors of this collection become undefined.
- Collection classes supporting Visual Builder send a `modifiedId` notification.

Exceptions

- `IOutOfMemory`
- `IFullException`, if the collection is bounded
- `IKeyAlreadyExistsException`, if the collection is a map or a sorted map



Bag

A *bag* is an unordered collection of zero or more elements with no key. Multiple elements are supported. A request to add an element that already exists is not ignored.

Figure 7 in the *Open Class Library User's Guide* gives an overview of the properties of a bag and its relationship to other flat collections.

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.

The following rule applies for duplicates: If bag P contains the element X *m* times and bag Q contains the element X *n* times, then the *union* of P and Q contains the element X *m+n* times, the *intersection* of P and Q contains the element X *MIN(m,n)* times, and the *difference* of P and Q contains the element X *m-n* times if *m* is > *n*, and *zero* times if *m* is ≤ *n*.

Derivation

Collection
  Equality Collection
    Bag

Variants and Header Files

`IBag`, the first class in the table below, is the default implementation variant. If you want to use polymorphism, you can replace the following class implementation variants by the reference class.

To use Visual Builder features with your collections, change the name of the desired collection class template in the list below from `I...` to `IV...`, and use the `ivbag.h` header file instead of the header file that you would normally use without Visual Builder.

Bag

Class Name	Header File	Implementation Variant
`IBag`	`ibag.h`	AVL tree
`IGBag`	`ibag.h`	AVL tree
`IBagOnBSTKeySortedSet`	`ibagbst.h`	B* tree
`IGBagOnBSTKeySortedSet`	`ibagbst.h`	B* tree
`IBagOnSortedLinkedSequence`	`ibagsls.h`	Linked Sequence
`IGBagOnSortedLinkedSequence`	`ibagsls.h`	Linked Sequence
`IBagOnSortedTabularSequence`	`ibagsts.h`	Tabular Sequence
`IGBagOnSortedTabularSequence`	`ibagsts.h`	Tabular Sequence
`IBagOnSortedDilutedSequence`	`ibagsds.h`	Diluted Sequence
`IGBagOnSortedDilutedSequence`	`ibagsds.h`	Diluted Sequence
`IBagOnHashKeySet`	`ibaghks.h`	Hash Table
`IGBagOnHashKeySet`	`ibaghks.h`	Hash Table

Members All member functions of flat collections are described in "Introduction to Flat Collections" on page 97. The following members are provided for bag:

Method	Page	Method	Page
Constructor	101	isEmpty	117
Copy Constructor	101	isFull	118
Destructor	101	locate	118
operator!=	102	locateNext	120
operator=	102	locateOrAdd	121
operator==	102	maxNumberOfElements	123
add	103	newCursor	123
addAllFrom	104	numberOfDifferentElements	123
addDifference	107	numberOfElements	123
addIntersection	108	numberOfOccurrences	124
addUnion	110	remove	125
allElementsDo	110	removeAllOccurrences	126
anyElement	112	removeAll	125
contains	113	removeAt	126
containsAllFrom	113	replaceAt	128
differenceWith	114	setToFirst	129
elementAt	115	setToNext	130
intersectionWith	117	setToNextDifferentElement	130
isBounded	117	unionWith	132

Bag also defines a cursor that inherits from `IElementCursor`. The members for `IElementCursor` are described in "Cursor" on page 267.

Template Arguments and Required Functions

Bag

```
IBag  <Element>
IGBag <Element, ECOps>
```

The default implementation of the class `IBag` requires the following element functions:

Element Type

- Default constructor
- Copy constructor
- Destructor
- Assignment
- Equality test
- Ordering relation

Bag on B* Key Sorted Set

```
IBagOnBSTKeySortedSet  <Element>
IGBagOnBSTKeySortedSet <Element, ECOps>
```

The default implementation of the class `IBagOnBSTKeySortedSet` requires the following element functions:

Element Type

- Default constructor
- Copy constructor
- Destructor
- Assignment
- Equality test
- Ordering relation

Bag on Sorted Linked Sequence

```
IBagOnSortedLinkedSequence  <Element>
IGBagOnSortedLinkedSequence <Element, ECOps>
```

The implementation of the class `IBagOnSortedLinkedSequence` requires the following element functions:

Element Type

- Default constructor
- Copy constructor
- Destructor
- Assignment
- Equality test
- Ordering relation

Bag on Sorted Tabular Sequence

`IBagOnSortedTabularSequence <`*`Element`*`>`
`IGBagOnSortedTabularSequence <`*`Element`*`, `*`ECOps`*`>`

The implementation of the class `IBagOnSortedTabularSequence` requires the following element functions:

Element Type

- Default constructor
- Copy constructor
- Destructor
- Assignment
- Equality test
- Ordering relation

Bag on Sorted Diluted Sequence

`IBagOnSortedDilutedSequence <`*`Element`*`>`
`IGBagOnSortedDilutedSequence <`*`Element`*`, `*`ECOps`*`>`

The implementation of the class `IBagOnSortedDilutedSequence` requires the following element functions:

Element Type

- Default constructor
- Copy constructor
- Destructor
- Assignment
- Equality test
- Ordering relation

Bag on Hash Key Set

```
IBagOnHashKeySet  <Element>
IGBagOnHashKeySet <Element, EHOps>
```

The implementation of the class `IBagOnHashKeySet` requires the following element functions:

Element Type

- Default constructor
- Copy constructor
- Destructor
- Assignment
- Equality test
- Hash function

Abstract Class and Reference Class

For polymorphism, you can use the corresponding abstract class, `IABag`, which is found in the `iabag.h` header file, or the corresponding reference class, `IRBag`, which is found in the `irbag.h` header file. See Chapter 11, “Polymorphic Use of Collections” in the *Open Class Library User's Guide* for further information.

Template Arguments and Required Functions

```
IABag <Element>
IRBag <Element, ConcreteBase>
```

The concrete base class is one of the classes defined above.

The required functions are the same as the required functions of the concrete base class.

Bag

Deque

A *deque* or double-ended queue is a sequence with restricted access. It is an ordered collection of elements with no key and no element equality. The elements are arranged so that each collection has a first and a last element, each element except the last has a next element, and each element but the first has a previous element. You can only add or remove the first or last element.

The type and value of the elements are irrelevant, and have no effect on the behavior of the collection.

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.

Derivation

Collection
  Ordered Collection
    Sequential Collection
      Sequence
        Deque

Note that deque is based on sequence but is not actually derived from it or from the other classes shown above. See "Restricted Access" in the *Open Class Library User's Guide* for further details.

Variants and Header Files

`IDeque`, the first class in the table below, is the default implementation variant. If you want to use polymorphism, you can replace the following class implementation variants by the reference class.

To use Visual Builder features with your collections, change the name of the desired collection class template in the list below from `I...` to `IV...`, and use the `ivdeque.h` header file instead of the header file that you would normally use without Visual Builder.

Deque

Class Name	Header File	Implementation Variant
`IDeque`	`ideque.h`	Linked sequence
`IGDeque`	`ideque.h`	Linked sequence
`IDequeOnTabularSequence`	`idquts.h`	Tabular sequence
`IGDequeOnTabularSequence`	`idquts.h`	Tabular sequence
`IDequeOnDilutedSequence`	`idquds.h`	Diluted sequence
`IGDequeOnDilutedSequence`	`idquds.h`	Diluted sequence

Members All members of flat collections are described in "Introduction to Flat Collections" on page 97. The following members are provided for deque:

Method	Page	Method	Page
Constructor	101	isFirst	117
Copy Constructor	101	isFull	118
Destructor	101	isLast	118
operator=	102	lastElement	118
add	103	maxNumberOfElements	123
addAllFrom	104	newCursor	123
addAsFirst	105	numberOfElements	123
addAsLast	105	removeAll	125
allElementsDo	110	removeFirst	127
anyElement	112	removeLast	128
compare	112	setToFirst	129
elementAt	115	setToLast	129
elementAtPosition	115	setToNext	130
firstElement	117	setToPosition	131
isBounded	117	setToPrevious	131
isEmpty	117		

Deque also defines a cursor that inherits from `IOrderedCursor`. The members for `IOrderedCursor` are described in "Cursor" on page 267.

Template Arguments and Required Functions

Deque

```
IDeque  <Element>
IGDeque <Element, StdOps>
```

The default implementation of the class `IDeque` requires the following element functions:

Element Type

- Copy constructor
- Destructor
- Assignment

Deque on Tabular Sequence

```
IDequeOnTabularSequence  <Element>
IGDequeOnTabularSequence <Element, StdOps>
```

The implementation of the class `IDequeOnTabularSequence` requires the following element functions:

Element Type

- Default constructor
- Copy constructor
- Destructor
- Assignment

Deque on Diluted Sequence

```
IDequeOnDilutedSequence  <Element>
IGDequeOnDilutedSequence <Element, StdOps>
```

The implementation of the class `IDequeOnDilutedSequence` requires the following element functions:

Element Type

- Default constructor
- copy constructor
- Assignment

Abstract Class and Reference Class

For polymorphism, you can use the corresponding abstract class, `IADeque`, which is found in the `iadeque.h` header file, or the corresponding reference class, `IRDeque`, which is found in the `irdeque.h` header file. See Chapter 11, "Polymorphic Use of Collections" in the *Open Class Library User's Guide* for further information.

Template Arguments and Required Functions

```
IADeque <Element>
IRDeque <Element, ConcreteBase>
```

The concrete base class is one of the classes defined above.

The required functions are the same as the required functions of the concrete base class.

Coding Example for Deque

The following program uses the default deque class, `IDeque`, to create a deque. It fills the deque with characters by adding them to the back end. The program then removes the characters from alternating ends of the deque (beginning with the front end) until the deque is empty.

The program uses the constant iterator class, `IConstantIterator`, when printing the collection. It uses the `addAsLast()` function to fill the deque and the `numberOfElements()` function to determine the deque's size. It uses the functions `firstElement()`, `removeFirst()`, `lastElement()`, and `removeLast()` to empty the deque.

```
//    letterdq.C  -  An example of using a Deque.

#include <iostream.h>

#include <ideque.h>
                     // Let's use the default deque
typedef IDeque <char> Deque;
                     // The deque requires iteration to be const
typedef IConstantIterator <char> CharIterator;

class Print : public CharIterator
{
public:
   IBoolean applyTo(char const&c)
      {
      cout << c;
      return True;
      }
};

/*-------------------------------------------------------------*\
| Test variables                                                |
\*-------------------------------------------------------------*/

char *String = "Teqikbonfxjmsoe  aydg.o zlarv pu o wr cu h";
```

```
/*-------------------------------------------------------------*\
| Main program                                                  |
\*-------------------------------------------------------------*/
int main()
{
   Deque D;
   char  C;
   IBoolean ReadFront = True;

   int i;

   // Put all characters in the deque.
   // Then read it, changing the end to read from
   // with every character read.

   cout << endl
        << "Adding characters to the back end of the deque:" << endl;

   for (i = 0; String[i] != 0; i ++) {
      D.addAsLast(String[i]);
      cout << String[i];
      }

   cout << endl << endl
        << "Current number of elements in the deque: "
        <<  D.numberOfElements() << endl;

   cout << endl
        << "Contents of the deque:" << endl;
   Print Aprinter;
   D.allElementsDo(Aprinter);

   cout << endl << endl
        << "Reading from the deque one element from front, one "
        << "from back, and so on:" << endl;

   while (!D.isEmpty())
      {
      if (ReadFront)                 // Read from front of Deque
         {
         C = D.firstElement();       // Get the character
         D.removeFirst();            // Delete it from the Deque
         }
      else
         {
         C = D.lastElement();
         D.removeLast();
         }
      cout << C;

      ReadFront = !ReadFront;     // Switch to other end of Deque
      }

   cout << endl;

   return(0);
}
```

The program produces the following output:

```
Adding characters to the back end of the deque:
Teqikbonfxjme vralz o.gdya  eospu o wr cu h

Current number of elements in the deque: 43

Contents of the deque:
Teqikbonfxjme vralz o.gdya  eospu o wr cu h

Reading from the deque one element from front, one from back, and so on:
The quick brown fox jumpes over a lazy dog.
```

Equality Sequence

An *equality sequence* is an ordered collection of elements. The elements are arranged so that each collection has a first and a last element, each element except the last has a next element, and each element but the first has a previous element. An equality sequence supports element equality, which gives you the ability, for example, to search for particular elements.

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.

Derivation

Collection
    Equality Collection
    Sequential Collection
        Equality Sequence

Figure 7 in the *Open Class Library User's Guide* illustrates the properties of an equality sequence and its relationship to other flat collections.

Variants and Header Files

`IEqualitySequence`, the first class in the table below, is the default implementation variant. If you want to use polymorphism, you can replace the following class implementation variants by the reference class.

To use Visual Builder features with your collections, change the name of the desired collection class template in the list below from `I...` to `IV...`, and use the `iveqseq.h` header file instead of the header file that you would normally use without Visual Builder.

Equality Sequence

Class Name	Header File	Implementation Variant
`IEqualitySequence`	`ieqseq.h`	Linked sequence
`IGEqualitySequence`	`ieqseq.h`	Linked sequence
`IEqualitySequenceOnTabularSequence`	`ieqts.h`	Tabular sequence
`IGEqualitySequenceOnTabularSequence`	`ieqts.h`	Tabular sequence
`IEqualitySequenceOnDilutedSequence`	`ieqds.h`	Diluted sequence
`IGEqualitySequenceOnDilutedSequence`	`ieqds.h`	Diluted sequence

Members All members of flat collections are described in “Introduction to Flat Collections” on page 97. The following members are provided for equality sequence:

Method	Page	Method	Page
Constructor	101	lastElement	118
Copy Constructor	101	locate	118
Destructor	101	locateFirst	119
operator!=	102	locateLast	119
operator=	102	locateNext	120
operator==	102	locateOrAdd	121
add	103	locatePrevious	122
addAllFrom	104	maxNumberOfElements	123
addAsFirst	105	newCursor	123
addAsLast	105	numberOfElements	123
addAsNext	106	numberOfOccurrences	124
addAsPrevious	106	remove	125
addAtPosition	107	removeAll	125
allElementsDo	110	removeAllOccurrences	126
anyElement	112	removeAt	126
compare	112	removeAtPosition	127
contains	113	removeFirst	127
containsAllFrom	113	removeLast	128
elementAt	115	replaceAt	128
elementAtPosition	115	setToFirst	129
firstElement	117	setToLast	129
isBounded	117	setToNext	130
isEmpty	117	setToPosition	131
isFirst	117	setToPrevious	131
isFull	118	sort	131
isLast	118		

Equality sequence also defines a cursor that inherits from `IOrderedCursor`. The members for `IOrderedCursor` are described in “Cursor” on page 267.

Template Arguments and Required Functions

Equality Sequence

```
IEqualitySequence  <Element>
IGEqualitySequence <Element, EOps>
```

The default implementation of `IEqualitySequence` requires the following element functions:

Element Type

- Assignment
- Equality test

Equality Sequence on Tabular Sequence

```
IEqualitySequenceOnTabularSequence  <Element>
IGEqualitySequenceOnTabularSequence <Element, EOps>
```

The implementation of the class `IEqualitySequenceOnTabularSequence` requires the following element functions:

Element Type

- Default constructor
- Copy constructor
- Destructor
- Assignment
- Equality test

Equality Sequence on Diluted Sequence

```
IEqualitySequenceOnDilutedSequence  <Element>
IGEqualitySequenceOnDilutedSequence <Element, EOps>
```

The implementation of the class `IEqualitySequenceOnDilutedSequence` requires the following element functions:

Element Type

- Default constructor
- Copy constructor
- Destructor

- Assignment
- Equality test

Abstract Class and Reference Class

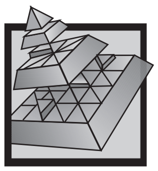
For polymorphism, you can use the corresponding abstract class, `IAEqSeq`, which is found in the `iaeqseq.h` header file, or the corresponding reference class, `IREqSeq`, which is found in the `ireqseq.h` header file. 📖 See Chapter 11, “Polymorphic Use of Collections” in the *Open Class Library User's Guide* for further information.

Template Arguments and Required Functions

```
IAEqSequ <Element>
IREqSequ <Element, ConcreteBase>
```

The concrete base class is one of the classes defined above.

The required functions are the same as the required functions of the concrete base class.



Heap

A *heap* is an unordered collection of zero or more elements with no key. Element equality is not supported. Multiple elements are supported. The type and value of the elements are irrelevant, and have no effect on the behavior of the 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.

Figure 7 in the *Open Class Library User's Guide* illustrates the properties of a heap and its relationship to other flat collections.

Derivation

Collection
  Heap

Variants and Header Files

`IHeap`, the first class in the table below, is the default implementation variant. If you want to use polymorphism, you can replace the following class implementation variants by the reference class.

To use Visual Builder features with your collections, change the name of the desired collection class template in the list below from `I...` to `IV...`, and use the `ivheap.h` header file instead of the header file that you would normally use without Visual Builder.

Class Name	Header File	Implementation Variant
`IHeap`	`iheap.h`	Linked sequence
`IGHeap`	`iheap.h`	Linked sequence
`IHeapOnTabularSequence`	`iheapts.h`	Tabular sequence
`IGHeapOnTabularSequence`	`iheapts.h`	Tabular sequence
`IHeapOnDilutedSequence`	`iheapds.h`	Diluted sequence
`IGHeapOnDilutedSequence`	`iheapds.h`	Diluted sequence

Heap

Members All members of flat collections are described in "Introduction to Flat Collections" on page 97. The following members are provided for heap:

Method	Page	Method	Page
Constructor	101	isEmpty	117
Copy Constructor	101	isFull	118
Destructor	101	maxNumberOfElements	123
operator=	102	newCursor	123
add	103	numberOfElements	123
addAllFrom	104	removeAll	125
allElementsDo	110	removeAt	126
anyElement	112	replaceAt	128
elementAt	115	setToFirst	129
isBounded	117	setToNext	130

Heap also defines a cursor that inherits from `IElementCursor`. The members for `IElementCursor` are described in "Cursor" on page 267.

Template Arguments and Required Functions

Heap

```
IHeap  <Element>
IGHeap <Element, StdOps>
```

The default implementation of `IHeap` requires the following element functions:

Element Type

- Copy constructor
- Assignment

Heap on Tabular Sequence

```
IHeapOnTabularSequence  <Element>
IGHeapOnTabularSequence <Element, StdOps>
```

The implementation of the class `IHeapOnTabularSequence` requires the following element functions:

Element Type

- Default constructor
- Copy constructor
- Assignment

Heap on Diluted Sequence

```
IHeapOnDilutedSequence  <Element>
IGHeapOnDilutedSequence <Element, StdOps>
```

The implementation of the class `IHeapOnDilutedSequence` requires the following element functions:

Element Type

- Default constructor
- Copy constructor

Abstract Class and Reference Class

For polymorphism, you can use the corresponding abstract class, `IAHeap`, which is found in the `iaheap.h` header file, or the corresponding reference class, `IRHeap`, which is found in the `irheap.h` header file. See Chapter 11, “Polymorphic Use of Collections” in the *Open Class Library User's Guide* for further information.

Template Arguments and Required Functions

```
IAHeap <Element>
IRHeap <Element, ConcreteBase>
```

The concrete base class is one of the heap classes.

The required functions are the same as the required functions of the concrete base class.

Coding Example for Heap

See “Coding Example for Key Sorted Set” on page 176 for an example of using a heap.

Heap

Key Bag

A *key bag* is an unordered collection of zero or more elements that have a key. Multiple elements are supported.

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 each element whose key matches your lock's serial number, and deletes one such element that has your name associated with it.

Figure 8 in the *Open Class Library User's Guide* illustrates the differences in behavior between map, relation, key set, and key bag when identical elements and elements with the same key are added.

Derivation

Collection
  Key Collection
    Key Bag

Figure 7 in the *Open Class Library User's Guide* gives an overview of the properties of a key bag and its relationship to other flat collections.

Variants and Header Files

`IKeyBag`, the first class in the table below, is the default implementation variant. If you want to use polymorphism, you can replace the following class implementation variants by the reference class.

To use Visual Builder features with your collections, change the name of the desired collection class template in the list below from `I...` to `IV...`, and use the `ivkeybag.h` header file instead of the header file that you would normally use without Visual Builder.

Key Bag

Class Name	Header File	Implementation Variant
`IKeyBag`	`ikeybag.h`	Hash table
`IGKeyBag`	`ikeybag.h`	Hash table
`IHashKeyBag`	`ihshkb.h`	Hash table
`IGHashKeyBag`	`ihshkb.h`	Hash table

Members All members of flat collections are described in "Introduction to Flat Collections" on page 97. The following members are provided for key bag:

Method	Page	Method	Page
Constructor	101	locateElementWithKey	119
Copy Constructor	101	locateNextElementWithKey	120
Destructor	101	locateOrAddElementWithKey	122
operator=	102	maxNumberOfElements	123
add	103	newCursor	123
addAllFrom	104	numberOfDifferentKeys	123
addOrReplaceElementWithKey	109	numberOfElements	123
allElementsDo	110	numberOfElementsWithKey	123
anyElement	112	removeAll	125
containsAllKeysFrom	113	removeAllElementsWithKey	126
containsElementWithKey	113	removeAt	126
elementAt	115	removeElementWithKey	127
elementWithKey	115	replaceAt	128
isBounded	117	replaceElementWithKey	129
isEmpty	117	setToFirst	129
isFull	118	setToNext	130
key	118	setToNextWithDifferentKey	130

Key Bag also defines a cursor that inherits from `IElementCursor`. The members for `IElementCursor` are described in "Cursor" on page 267.

Template Arguments and Required Functions

Key Bag

```
IKeyBag  <Element, Key>
IGKeyBag <Element, Key, KEHOps>
```

The default implementation of the class `IKeyBag` requires the following element and key-type functions:

Element Type

- Copy constructor
- Destructor
- Assignment
- Key access

Key Type

- Equality test
- Hash function

Hash Key Bag

```
IHashKeyBag  <Element, Key>
IGHashKeyBag <Element, Key, KEHOps>
```

The implementation of the class `IHashKeyBag` requires the following element and key-type functions:

Element Type

- Copy constructor
- Destructor
- Assignment
- Key access

Key Type

- Equality test
- Hash function

Abstract Class and Reference Class

For polymorphism, you can use the corresponding abstract class, `IAKeyBag`, which is found in the `iakeybag.h` header file, or the corresponding reference class, `IRKeyBag`, which is found in the `irkeybag.h` header file. 📖 See Chapter 11, "Polymorphic Use of Collections" in the *Open Class Library User's Guide* for further information.

Template Arguments and Required Functions

```
IAKeyBag <Element, Key>
IRKeyBag <Element, Key, ConcreteBase>
```

The concrete base class is one of the classes defined above.

The required functions are the same as the required functions of the concrete base class.

Coding Example for Key Bag

The following program uses the default key bag class, IKeyBag, to create a key bag for storing observations made on animals. The key of the class is the name of the animal. The program produces various reports regarding the observations. Then it removes all the extinct animals, which are stored in a sequence, from the key bag.

The program uses the add() function to fill the key bag and the forCursor macro to display the observations. It uses the following functions to produce the reports:

- numberOfElements()
- numberOfDifferentKeys()
- numberOfElementsWithKey()
- locateElementWithKey()
- setToNextElementWithKey()
- removeAllElementsWithKey()

See Appendix A, "Header Files for Collection Class Library Coding Examples" on page 575 for the code of the animal.h file.

```
// animals.C  -  An example of using a Key Bag
#include <iostream.h>
               // Class Animal:
#include "animal.h"

               // Let's use the default Key Bag:
#include <ikeybag.h>
typedef IKeyBag<Animal, IString> Animals;

               // For keys let's use the default Sequence:
#include <iseq.h>
typedef ISequence<IString> Names;


main() {

   Animals animals;
   Animals::Cursor animalsCur1(animals), animalsCur2(animals);

   animals.add(Animal("bear", "heavy"));
   animals.add(Animal("bear", "strong"));
   animals.add(Animal("dinosaur", "heavy"));
   animals.add(Animal("dinosaur", "huge"));
   animals.add(Animal("dinosaur", "extinct"));
   animals.add(Animal("eagle", "black"));
   animals.add(Animal("eagle", "strong"));
   animals.add(Animal("lion", "dangerous"));
   animals.add(Animal("lion", "strong"));
   animals.add(Animal("mammoth", "long haired"));
   animals.add(Animal("mammoth", "extinct"));
   animals.add(Animal("sabre tooth tiger", "extinct"));
   animals.add(Animal("zebra", "striped"));

                 // Display all elements in animals:
   cout << endl
        << "All our observations on animals:" << endl;
   forCursor(animalsCur1)  cout << "    " << animalsCur1.element();
   cout << endl << endl
        << "Number of observations on animals: "
        << animals.numberOfElements() << endl;

   cout << endl
        << "Number of different animals: "
        << animals.numberOfDifferentKeys() << endl;

   Names namesOfExtinct(animals.numberOfDifferentKeys());
   Names::Cursor extinctCur1(namesOfExtinct);

   animalsCur1.setToFirst();
   do {
      IString name = animalsCur1.element().name();

      cout << endl
           << "We have " << animals.numberOfElementsWithKey(name)
           << " observations on " << name << ":" << endl;

                  // We need to use a separate cursor here
                  // because otherwise animalsCur1 would become
                  // invalid after last locateNextElement...()
      animals.locateElementWithKey(name, animalsCur2);
      do  {
         IString attribute = animalsCur2.element().attribute();
         cout << "    " << attribute << endl;
         if (attribute == "extinct") namesOfExtinct.add(name);
      } while (animals.locateNextElementWithKey(name, animalsCur2));

   } while (animals.setToNextWithDifferentKey(animalsCur1));

                // Remove all observations on extinct animals:
   forCursor(extinctCur1)
      animals.removeAllElementsWithKey(extinctCur1.element());

                // Display all elements in animals:
   cout << endl << endl
        << "After removing all observations on extinct animals:" << endl;
   forCursor(animalsCur1)  cout << "    " << animalsCur1.element();

   cout << endl
        << "Number of observations on animals: "
        << animals.numberOfElements() << endl;

   cout << endl
        << "Number of different animals: "
        << animals.numberOfDifferentKeys() << endl;

   return 0;
}
```

The program produces the following output:

```
All our observations on animals:
    The eagle is strong.
    The eagle is black.
    The bear is strong.
    The bear is heavy.
    The zebra is striped.
    The mammoth is extinct.
    The mammoth is long haired.
    The lion is strong.
    The lion is dangerous.
    The dinosaur is extinct.
    The dinosaur is huge.
    The dinosaur is heavy.
    The sabre tooth tiger is extinct.


Number of observations on animals: 13

Number of different animals: 7

We have 2 observations on eagle:
    strong
    black

We have 2 observations on bear:
    strong
    heavy

We have 1 observations on zebra:
    striped

We have 2 observations on mammoth:
    extinct
    long haired

We have 2 observations on lion:
    strong
    dangerous

We have 3 observations on dinosaur:
    extinct
    huge
    heavy

We have 1 observations on sabre tooth tiger:
    extinct


After removing all observations on extinct animals:
    The eagle is strong.
    The eagle is black.
    The bear is strong.
    The bear is heavy.
    The zebra is striped.
    The lion is strong.
    The lion is dangerous.

Number of observations on animals: 7

Number of different animals: 4
```

Key Set

A *key set* is an unordered collection of zero or more elements that have a key. Element equality is not supported. Only unique elements are supported, in terms of their key.

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.

Figure 8 in the *Open Class Library User's Guide* illustrates the differences in behavior between map, relation, key set, and key bag when identical elements and elements with the same key are added.

Figure 7 in the *Open Class Library User's Guide* gives an overview of the properties of a key set and its relationship to other flat collections.

Derivation

Collection
  Key Collection
    Key Set

Variants and Header Files

`IKeySet`, the first class in the table below, is the default implementation variant. If you want to use polymorphism, you can replace the following class implementation variants by the reference class.

To use Visual Builder features with your collections, change the name of the desired collection class template in the list below from `I...` to `IV...`, and use the `ivkeyset.h` header file instead of the header file that you would normally use without Visual Builder.

Class Name	Header File	Implementation Variant
`IKeySet`	`ikeyset.h`	AVL tree
`IGKeySet`	`ikeyset.h`	AVL tree
`IKeySetOnBSTKeySortedSet`	`iksbst.h`	B* tree
`IGKeySetOnBSTKeySortedSet`	`iksbst.h`	B* tree

Key Set

Class Name	Header File	Implementation Variant
`IHashKeySet`	`ihshks.h`	Hash table
`IGHashKeySet`	`ihshks.h`	Hash table
`IKeySetOnSortedLinkedSequence`	`ikssls.h`	Linked sequence
`IGKeySetOnSortedLinkedSequence`	`ikssls.h`	Linked sequence
`IKeySetOnSortedTabularSequence`	`ikssts.h`	Tabular sequence
`IGKeySetOnSortedTabularSequence`	`ikssts.h`	Tabular sequence
`IKeySetOnSortedDilutedSequence`	`ikssds.h`	Diluted sequence
`IGKeySetOnSortedDilutedSequence`	`ikssds.h`	Diluted sequence

Members All members of flat collections are described in "Introduction to Flat Collections" on page 97. The following members are provided for key set:

Method	Page	Method	Page
Constructor	101	isFull	118
Copy Constructor	101	key	118
Destructor	101	locateElementWithKey	119
operator=	102	locateOrAddElementWithKey	122
add	103	maxNumberOfElements	123
addAllFrom	104	newCursor	123
addOrReplaceElementWithKey	109	numberOfElements	123
allElementsDo	110	removeAll	125
anyElement	112	removeAt	126
containsAllKeysFrom	113	removeElementWithKey	127
containsElementWithKey	113	replaceAt	128
elementAt	115	replaceElementWithKey	129
elementWithKey	115	setToFirst	129
isBounded	117	setToNext	130
isEmpty	117		

Key set also defines a cursor that inherits from `IElementCursor`. The members for `IElementCursor` are described in "Cursor" on page 267.

Template Arguments and Required Functions

Key Set

```
IKeySet  <Element, Key>
IGKeySet <Element, Key, KCOps>
```

The default implementation of the class `IKeySet` requires the following element and key-type functions:

Element Type

- Copy constructor
- Destructor
- Assignment
- Key access

Key Type

Ordering relation

Key Set on B* Key Sorted Set

```
IKeySetOnBSTKeySortedSet  <Element, Key>
IGKeySetOnBSTKeySortedSet <Element, Key, KCOps>
```

The implementation of the class `IKeySetOnBSTKeySortedSet` requires the following element and key-type functions:

Element Type

- Default constructor
- Copy constructor
- Destructor
- Assignment
- Key access

Key Type

Ordering relation

Hash Key Set

```
IHashKeySet  <Element, Key>
IGHashKeySet <Element, Key, KEHOps>
```

The implementation class `IHashKeySet` requires the following element and key-type functions:

Element Type

- Copy constructor
- Destructor
- Assignment
- Key access

Key Set

Key Type

- Equality test
- Hash function

Key Set on Sorted Linked Sequence

`IKeySetOnSortedLinkedSequence  <`*`Element`*`, `*`Key`*`>`
`IGKeySetOnSortedLinkedSequence <`*`Element`*`, `*`Key`*`, `*`KCOps`*`>`

The implementation of the class `IKeySetOnSortedLinkedSequence` requires the following element and key-type functions:

Element Type

- Copy constructor
- Destructor
- Assignment
- Key access

Key Type

Ordering relation

Key Set on Sorted Tabular Sequence

`IKeySetOnSortedTabularSequence  <`*`Element`*`, `*`Key`*`>`
`IGKeySetOnSortedTabularSequence <`*`Element`*`, `*`Key`*`, `*`KCOps`*`>`

The implementation of the class `IKeySetOnSortedTabularSequence` requires the following element and key-type functions:

Element Type

- Default constructor
- Copy constructor
- Destructor
- Assignment
- Key access

Key Type

Ordering relation

Key Set on Sorted Diluted Sequence

`IKeySetOnSortedDilutedSequence` *<Element, Key>*
`IGKeySetOnSortedDilutedSequence` *<Element, Key, KCOps>*

The implementation of the class `IKeySetOnSortedDilutedSequence` requires the following element and key-type functions:

Element Type

- Default constructor
- Copy constructor
- Destructor
- Assignment
- Key access

Key Type

Ordering relation

Abstract Class and Reference Class

For polymorphism, you can use the corresponding abstract class, `IAKeySet`, which is found in the `iakeyset.h` header file, or the corresponding reference class, `IRKeySet`, which is found in the `irkeyset.h` header file. See Chapter 11, "Polymorphic Use of Collections" in the *Open Class Library User's Guide* for further information.

Template Arguments and Required Functions

`IAKeySet` *<Element, Key>*
`IRKeySet` *<Element, Key, ConcreteBase>*

The concrete base class is one of the classes defined above.

The required functions are the same as the required functions of the concrete base class.

Coding Example for Key Set

The following program implements a key set using the default class, `IKeySet`. The program adds four elements to the key set and then removes one element by looking for a key. If an exception occurs, it displays the exception name and description.

The program uses cursor iteration (the `forCursor` macro) to display the contents of the collection. To add and remove elements, it uses the `add()` function and the `removeElementWithKey()` function.

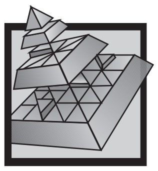
Key Set

See Appendix A, “Header Files for Collection Class Library Coding Examples” on page 575 for the code of the `demoelem.h` file.

```
//  intkyset.C  -  An example of using a Key Set
   #include <iostream.h>
   #include <iglobals.h>
   #include <icursor.h>

   #include <ikeyset.h>
                         // Class DemoElement:
   #include "demoelem.h"

   typedef IKeySet < DemoElement,int > TestKeySet;

   ostream & operator<< ( ostream & sout, TestKeySet const & t){
     sout << t.numberOfElements() << " elements are in the set:\n";

     TestKeySet::Cursor cursor (t);
     //  forCursor(c)
     // expands to
     //  for ((c).setToFirst (); (c).isValid (); (c).setToNext ())

     forCursor (cursor)
       sout << "   " << cursor.element() << "\n";
     return sout << "\n";
   }

   main(){
     TestKeySet t;
    try {
        t.add(DemoElement(1,1));
        t.add(DemoElement(2,4711));
        t.add(DemoElement(3,1));
        t.add(DemoElement(4,443));

        cout << t;

        t.removeElementWithKey (3);
        cout << t;
      }
    catch (IException & exception) {
      cout << exception.name() << " : " << exception.text();
      }

     return 0;
   }
```

The program produces the following output:

```
   4 elements are in the set:
      1,1
      2,4711
      3,1
      4,443

   3 elements are in the set:
      1,1
      2,4711
      4,443
```



Key Sorted Bag

A *key sorted bag* is an ordered collection of zero or more elements that have a key. Elements are sorted according to the value of their key field. Element equality is not supported. Multiple elements are supported.

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.

Figure 7 in the *Open Class Library User's Guide* gives an overview of the properties of a key sorted bag and its relationship to other flat collections.

Derivation

```
          Collection
               Ordered Collection
Key Collection          Sorted Collection
      Key Sorted Collection
         Key Sorted Bag
```

Variants and Header Files

IKeySortedBag, the first class in the table below, is the default implementation variant. If you want to use polymorphism, you can replace the following class implementation variants by the reference class.

To use Visual Builder features with your collections, change the name of the desired collection class template in the list below from `I...` to `IV...`, and use the `ivksbag.h` header file instead of the header file that you would normally use without Visual Builder.

Class Name	Header File	Implementation Variant
IKeySortedBag	iksbag.h	Linked sequence
IGKeySortedBag	iksbag.h	Linked sequence
IKeySortedBagOnSortedTabularSequence	iksbsts.h	Tabular sequence
IGKeySortedBagOnSortedTabularSequence	iksbsts.h	Tabular sequence
IKeySortedBagOnSortedDilutedSequence	iksbsds.h	Diluted sequence
IGKeySortedBagOnSortedDilutedSequence	iksbsds.h	Diluted sequence

Key Sorted Bag

Members All members of flat collections are described in "Introduction to Flat Collections" on page 97. The following members are provided for key sorted bag:

Method	Page	Method	Page
Constructor	101	locateElementWithKey	119
Copy Constructor	101	locateNextElementWithKey	120
Destructor	101	locateOrAddElementWithKey	122
operator=	102	maxNumberOfElements	123
add	103	newCursor	123
addAllFrom	104	numberOfDifferentKeys	123
addOrReplaceElementWithKey	109	numberOfElements	123
allElementsDo	110	numberOfElementsWithKey	123
anyElement	112	removeAll	125
compare	112	removeAllElementsWithKey	126
containsAllKeysFrom	113	removeAt	126
containsElementWithKey	113	removeAtPosition	127
elementAt	115	removeElementWithKey	127
elementAtPosition	115	removeFirst	127
elementWithKey	115	removeLast	128
firstElement	117	replaceAt	128
isBounded	117	replaceElementWithKey	129
isEmpty	117	setToFirst	129
isFirst	117	setToLast	129
isFull	118	setToNext	130
isLast	118	setToNextWithDifferentKey	130
key	118	setToPosition	131
lastElement	118	setToPrevious	131

Key sorted bag also defines a cursor that inherits from `IOrderedCursor`. The members for `IOrderedCursor` are described in "Cursor" on page 267.

Template Arguments and Required Functions

Key Sorted Bag

```
IKeySortedBag  <Element, Key>
IGKeySortedBag <Element, Key, KCOps>
```

The implementation of the class `IKeySortedBag` requires the following element and key-type functions:

Element Type

- Copy constructor
- Destructor
- Assignment
- Key access

Key Type

Ordering relation

Key Sorted Bag on Tabular Sequence

`IKeySortedBagOnTabularSequence` *<Element, Key>*
`IGKeySortedBagOnTabularSequence` *<Element, Key, KCOps>*

The implementation of the class `IKeySortedBagOnTabularSequence` requires the following element and key-type functions:

Element Type

- Default constructor
- Copy constructor
- Destructor
- Assignment
- Key access

Key Type

Ordering relation

Key Sorted Bag on Diluted Sequence

`IKeySortedBagOnDilutedSequence` *<Element, Key>*
`IGKeySortedBagOnDilutedSequence` *<Element, Key, KCOps>*

The implementation of the class `IKeySortedBagOnDilutedSequence` requires the following element and key-type functions:

Element Type

- Default constructor
- Copy constructor
- Destructor
- Assignment
- Key access

Key Type

Ordering relation

Abstract Class and Reference Class

For polymorphism, you can use the corresponding abstract class, `IAKeySortedBag`, which is found in the `iaksbag.h` header file, or the corresponding reference class, `IRKeySortedBag`, which is found in the `irksbag.h` header file. See Chapter 11, "Polymorphic Use of Collections" in the *Open Class Library User's Guide* for further information.

Template Arguments and Required Functions

`IAKSBag <`*`Element`*`, `*`Key`*`>`
`IRKSBag <`*`Element`*`, `*`Key`*`, `*`ConcreteBase`*`>`

The concrete base class is one of the classes defined above.

The required functions are the same as the required functions of the concrete base class.

Coding Example for Key Sorted Bag

The following program illustrates the use of a key sorted bag. The program determines the number of words that have the same length in a phrase. It stores the words of the phrase in a key sorted bag that it implements using the default class, `IKeySortedBag`. The program makes the key the length of the word. Because of the properties of a key sorted bag, it sorts the words by their length (the key), and it stores all duplicate words.

The program determines the number of different word lengths using the `numberOfDifferentKeys()` function. It uses the `numberOfElementsWithKey()` function and the `setToNextWithDifferentKey()` function to iterate through the collection and count the number of words with the same length.

See Appendix A, "Header Files for Collection Class Library Coding Examples" on page 575 for the code of the `toyword.h` file.

```
// wordbag.C  -  An example of using a Key Sorted Bag
#include <iostream.h>
                             // Class Word
#include "toyword.h"
                             // Let's use the defaults:
#include <iksbag.h>

typedef IKeySortedBag <Word, unsigned> WordBag;


int main()
{
   IString    phrase[] = {"people", "who", "live", "in", "glass",
                  "houses", "should", "not", "throw", "stones"};
   const size_t phraseWords = sizeof(phrase) / sizeof(IString);

   WordBag wordbag(phraseWords);

   for (int cnt=0; cnt < phraseWords; cnt++)  {
    unsigned keyValue = phrase[cnt].length();
    Word theWord(phrase[cnt],keyValue);
    wordbag.add (theWord);
   }

   cout << "Contents of our WordBag sorted by number of letters:" << endl;

   WordBag::Cursor wordBagCursor(wordbag);
   forCursor(wordBagCursor)
     cout << "WB: " << wordBagCursor.element().getWord() << endl;

   cout << endl << "Our phrase has " << phraseWords << " words." << endl                       ;
   cout << "In our WordBag are " << wordbag.numberOfElements()
        << " words." << endl << endl;

   cout << "There are " << wordbag.numberOfDifferentKeys()
        << " different word lengths." << endl << endl;

   wordBagCursor.setToFirst();
   do  {
      unsigned letters = wordbag.key(wordBagCursor.element());
      cout << "There are "
          << wordbag.numberOfElementsWithKey(letters)
          << " words with " << letters << " letters." << endl;
   }  while  (wordbag.setToNextWithDifferentKey(wordBagCursor));

   return 0;
}
```

This program produces the following output:

```
Contents of our WordBag sorted by number of letters:
WB: in
WB: who
WB: not
WB: live
WB: glass
WB: throw
WB: people
WB: houses
WB: should
WB: stones

Our phrase has 10 words.
In our WordBag are 10 words.

There are 5 different word lengths.

There are 1 words with 2 letters.
There are 2 words with 3 letters.
There are 1 words with 4 letters.
There are 2 words with 5 letters.
There are 4 words with 6 letters.
```

Key Sorted Set

A *key sorted set* is an ordered collection of zero or more elements that have a key. Elements are sorted according to the value of their key field. Element equality is not supported. Only elements with unique keys are supported. A request to add an element whose key already exists is ignored.

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.

Figure 7 in the *Open Class Library User's Guide* gives an overview of the properties of a key sorted set and its relationship to other flat collections.

Derivation

```
                    Collection
                              Ordered Collection
   Key Collection                  Sorted Collection
              Key Sorted Collection
                 Key Sorted Set
```

Variants and Header Files

`IKeySortedSet`, the first class in the table below, is the default implementation variant. If you want to use polymorphism, you can replace the following class implementation variants by the reference class.

To use Visual Builder features with your collections, change the name of the desired collection class template in the list below from `I...` to `IV...`, and use the `ivksset.h` header file instead of the header file that you would normally use without Visual Builder.

Class Name	Header File	Implementation Variant
`IKeySortedSet`	`iksset.h`	AVL tree
`IGKeySortedSet`	`iksset.h`	AVL tree
`IAVLKeySortedSet`	`iavlkss.h`	AVL tree
`IGAVLKeySortedSet`	`iavlkss.h`	AVL tree

Key Sorted Set

Class Name	Header File	Implementation Variant
`IBSTKeySortedSet`	`ibstkss.h`	B* tree
`IGBSTKeySortedSet`	`ibstkss.h`	B* tree
`IKeySortedSetOnSortedLinkedSequence`	`iksssls.h`	Linked sequence
`IGKeySortedSetOnSortedLinkedSequence`	`iksssls.h`	Linked sequence
`IKeySortedSetOnSortedTabularSequence`	`iksssts.h`	Tabular sequence
`IGKeySortedSetOnSortedTabularSequence`	`iksssts.h`	Tabular sequence
`IKeySortedSetOnSortedDilutedSequence`	`iksssds.h`	Diluted sequence
`IGKeySortedSetOnSortedDilutedSequence`	`iksssds.h`	Diluted sequence

Members All members of flat collections are described in "Introduction to Flat Collections" on page 97. The following members are provided for key sorted set:

Method	Page	Method	Page
Constructor	101	key	118
Copy Constructor	101	lastElement	118
Destructor	101	locateElementWithKey	119
operator=	102	locateNextElementWithKey	120
add	103	locateOrAddElementWithKey	122
addAllFrom	104	maxNumberOfElements	123
addOrReplaceElementWithKey	109	newCursor	123
allElementsDo	110	numberOfElements	123
anyElement	112	removeAll	125
compare	112	removeAt	126
containsAllKeysFrom	113	removeAtPosition	127
containsElementWithKey	113	removeElementWithKey	127
elementAt	115	removeFirst	127
elementAtPosition	115	removeLast	128
elementWithKey	115	replaceAt	128
firstElement	117	replaceElementWithKey	129
isBounded	117	setToFirst	129
isEmpty	117	setToLast	129
isFirst	117	setToNext	130
isFull	118	setToPosition	131
isLast	118	setToPrevious	131

Key Sorted Set also defines a cursor that inherits from `IOrderedCursor`. The members for `IOrderedCursor` are described in "Cursor" on page 267.

Template Arguments and Required Functions

Key Sorted Set

```
IKeySortedSet  <Element, Key>
IGKeySortedSet <Element, Key, KCOps>
```

The implementation of the class `IKeySortedSet` requires the following element and key-type functions:

Element Type

- Copy constructor
- Destructor
- Assignment
- Key access

Key Type

Ordering relation

AVL Key Sorted Set

```
IAVLKeySortedSet  <Element>
IGAVLKeySortedSet <Element, Key, KCOps>
```

The implementation of the class `IAVLKeySortedSet` requires the following element and key-type functions:

Element Type

- Copy constructor
- Assignment
- Destructor
- Key access

Key Type

Ordering relation

B* Key Sorted Set

```
IBSTKeySortedSet  <Element, Key>
IBSTKeySortedSet  <Element, Key, KCOps>
```

The implementation of the class `IBSTKeySortedSet` requires the following element and key-type functions:

Key Sorted Set

Element Type

- Default constructor
- Copy constructor
- Destructor
- Assignment
- Key access

Key Type

Ordering relation

Key Sorted Set on Sorted Linked Sequence

```
IKeySortedSetOnSortedLinkedSequence  <Element>
IGKeySortedSetOnSortedLinkedSequence <Element, Key, KCOps>
```

The implementation of the class `IKeySortedSetOnSortedLinkedSequence` requires the following element and key-type functions:

Element Type

- Copy constructor
- Destructor
- Assignment
- Key access

Key Type

Ordering relation

Key Sorted Set on Sorted Tabular Sequence

```
IKeySortedSetOnSortedTabularSequence  <Element>
IGKeySortedSetOnSortedTabularSequence <Element, Key, KCOps>
```

The implementation of the class `IKeySortedSetOnSortedTabularSequence` requires the following element and key-type functions:

Element Type

- Default constructor
- Copy constructor
- Destructor
- Assignment
- Key access

Key Type

Ordering relation

Key Sorted Set on Sorted Diluted Sequence

```
IKeySortedSetOnSortedDilutedSequence  <Element, Key>
IGKeySortedSetOnSortedDilutedSequence <Element, Key, KCOps>
```

The implementation of the class `IKeySortedSetOnSortedDilutedSequence` requires the following element and key-type functions:

Element Type

- Default constructor
- Copy constructor
- Destructor
- Assignment
- Key access

Key Type

Ordering relation

Abstract Class and Reference Class

For polymorphism, you can use the corresponding abstract class, `IAKeySortedSet`, which is found in the `iaksset.h` header file, or the corresponding reference class, `IRKeySortedSet`, which is found in the `irksset.h` header file. See Chapter 11, "Polymorphic Use of Collections" in the *Open Class Library User's Guide* for further information.

Template Arguments and Required Functions

```
IAKeySortedSet <Element, Key>
IRKeySortedSet <Element, Key, ConcreteBase>
```

The concrete base class is one of the classes defined above.

The required functions are the same as the required functions of the concrete base class.

Coding Example for Key Sorted Set

The following program uses the default classes for a key sorted set and a heap, `IKeySortedSet` and `IHeap`, to track parcels for a delivery service. It uses a key sorted set to record the parcels that are currently in circulation. The fast access of a sorted collection is not necessary to keep track of the delivered parcels, so the program uses a heap to keep track of them.

The parcel element contains three data members: one of type `PlaceTime` that stores the origin time and place of the parcel, another of type `PlaceTime` that stores the current time and place of the parcel, and one of type `ToyString` that stores the destination.

The function `update()` adds parcels that have arrived at their destinations to the heap of delivered parcels, and removes them from the key sorted set for circulating parcels.

The program uses the `add()` function to update and the `removeAll()` function to remove elements from the key sorted set.

See Appendix A, "Header Files for Collection Class Library Coding Examples" on page 575 for the code of the `parcel.h` file.

```
// parcel.C  -  An example of using a Key Sorted Set and a Heap
#include <iostream.h>

#include "parcel.h"
                          // Let's use the default KeySorted Set:
#include <iksset.h>
                          // Let's use the default Heap:
#include <iheap.h>

typedef IKeySortedSet<Parcel, ToyString> ParcelSet;
typedef IHeap        <Parcel>            ParcelHeap;

ostream& operator<<(ostream&, ParcelSet const&);
ostream& operator<<(ostream&, ParcelHeap const&);

void update(ParcelSet&, ParcelHeap&);


main()  {

    ParcelSet circulating;
    ParcelHeap delivered;

    int today = 8;
```

```
   circulating.add(Parcel("London", "Athens",
      today,      "26LoAt"));
   circulating.add(Parcel("Amsterdam", "Toronto",
      today += 2, "27AmTo"));
   circulating.add(Parcel("Washington", "Stockholm",
      today += 5, "25WaSt"));
   circulating.add(Parcel("Dublin", "Kairo",
      today += 1, "25DuKa"));
   update(circulating, delivered);
   cout << "\nThe situation at start:\n";
   cout << "Parcels in circulation:\n" << circulating;

   today ++;
   circulating.elementWithKey("27AmTo").arrivedAt(
      "Atlanta",   today);
   circulating.elementWithKey("25WaSt").arrivedAt(
      "Amsterdam", today);
   circulating.elementWithKey("25DuKa").arrivedAt(
      "Paris",     today);
   update(circulating, delivered);
   cout << "\n\nThe situation at day " << today << ":\n";
   cout << "Parcels in circulation:\n" << circulating;

   today ++;          // One day later ...
   circulating.elementWithKey("27AmTo").arrivedAt("Chicago", today);
           // As in real life, one parcel gets lost:
   circulating.removeElementWithKey("26LoAt");
   update(circulating, delivered);
   cout << "\n\nThe situation at day " << today << ":\n";
   cout << "Parcels in circulation:\n" << circulating;

   today ++;
   circulating.elementWithKey("25WaSt").arrivedAt("Oslo", today);
   circulating.elementWithKey("25DuKa").arrivedAt("Kairo", today);
           // New parcels are shipped.
   circulating.add(Parcel("Dublin", "Rome", today,   "27DuRo"));
           // Let's try to add one with a key already there.
           // The KeySsorted Set should ignore it:
   circulating.add(Parcel("Nowhere", "Nirvana", today, "25WaSt"));
   update(circulating, delivered);
   cout << "\n\nThe situation at day " << today << ":\n";
   cout << "Parcels in circulation:\n" << circulating;
   cout << "Parcels delivered:\n" << delivered;

                 // Now we make all parcels arrive today:
   today ++;

   ParcelSet::Cursor circulatingcursor(circulating);
   forCursor(circulatingcursor) {
      circulating.elementAt(circulatingcursor).wasDelivered(today);
   }
   update(circulating, delivered);
   cout << "\n\nThe situation at day " << today << ":\n";
   cout << "Parcels in circulation:\n" << circulating;
   cout << "Parcels delivered:\n" << delivered;

   if (circulating.isEmpty())
      cout << "\nAll parcels were delivered.\n";
   else
      cout << "\nSomething very strange happened here.\n";

   return  0;
}
```

```
ostream& operator<<(ostream& os, ParcelSet const& parcels)  {
    ParcelSet::Cursor pcursor(parcels);
    forCursor(pcursor) {
       os <<  pcursor.element() << "\n";
    }
    return os;
}

ostream& operator<<(ostream& os, ParcelHeap const& parcels)  {
    ParcelHeap::Cursor pcursor(parcels);
    forCursor(pcursor) {
       os <<  pcursor.element() << "\n";
    }
    return os;
}

Boolean wasDelivered(Parcel const& p, void* dp) {
     if ( p.lastArrival().city() == p.destination() ) {
        ((ParcelHeap*)dp)->add(p);
        return True;
     }
     else
        return False;
}

void update(ParcelSet& p, ParcelHeap& d) {
     p.removeAll(wasDelivered, &d);
}
```

The program produces the following output:

```
The situation at start:
Parcels in circulation:
25DuKa: From Dublin(day 16) to Kairo
            is at Dublin  since day 16.
25WaSt: From Washington(day 15) to Stockholm
            is at Washington  since day 15.
26LoAt: From London(day 8) to Athens
            is at London  since day 8.
27AmTo: From Amsterdam(day 10) to Toronto
            is at Amsterdam  since day 10.


The situation at day 17:
Parcels in circulation:
25DuKa: From Dublin(day 16) to Kairo
            is at Paris  since day 17.
25WaSt: From Washington(day 15) to Stockholm
            is at Amsterdam  since day 17.
26LoAt: From London(day 8) to Athens
            is at London  since day 8.
27AmTo: From Amsterdam(day 10) to Toronto
            is at Atlanta  since day 17.


The situation at day 18:
Parcels in circulation:
25DuKa: From Dublin(day 16) to Kairo
            is at Paris  since day 17.
25WaSt: From Washington(day 15) to Stockholm
            is at Amsterdam  since day 17.
27AmTo: From Amsterdam(day 10) to Toronto
            is at Chicago  since day 18.
```

```
The situation at day 19:
Parcels in circulation:
25WaSt: From Washington(day 15) to Stockholm
            is at Oslo  since day 19.
27AmTo: From Amsterdam(day 10) to Toronto
            is at Chicago  since day 18.
27DuRo: From Dublin(day 19) to Rome
            is at Dublin  since day 19.
Parcels delivered:
25DuKa: From Dublin(day 16) to Kairo
            was delivered on day 19.


The situation at day 20:
Parcels in circulation:
Parcels delivered:
25DuKa: From Dublin(day 16) to Kairo
            was delivered on day 19.
25WaSt: From Washington(day 15) to Stockholm
            was delivered on day 20.
27AmTo: From Amsterdam(day 10) to Toronto
            was delivered on day 20.
27DuRo: From Dublin(day 19) to Rome
            was delivered on day 20.

All parcels were delivered.
```

Key Sorted Set

Map

A *map* is an unordered collection of zero or more elements that have a key. Element equality is supported and the values of the elements are relevant.

Only elements with unique keys are supported. A request to add an element whose key already exists in another element of the collection causes an exception to be thrown. A request to add a duplicate element is ignored.

An example of using a map is a program that translates integer values between the ranges of 0 and 20 to their written equivalents, or between written numbers and their numeric values. 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.

Figure 8 in the *Open Class Library User's Guide* illustrates the differences in behavior between map, relation, key set, and key bag when identical elements and elements with the same key are added.

Figure 7 in the *Open Class Library User's Guide* gives an overview of the properties of a map and its relationship to other flat collections.

Derivation

```
                Collection
Key Collection              Equality Collection
            Equality Key Collection
                    Map
```

Variants and Header Files

`IMap`, the first class in the table below, is the default implementation variant. If you want to use polymorphism, you can replace the following class implementation variants by the reference class.

To use Visual Builder features with your collections, change the name of the desired collection class template in the list below from `I...` to `IV...`, and use the `ivmap.h` header file instead of the header file that you would normally use without Visual Builder.

Map

Class Name	Header File	Implementation Variant
`IMap`	`imap.h`	AVL tree
`IGMap`	`imap.h`	AVL tree
`IMapOnBSTKeySortedSet`	`imapbst.h`	B* tree
`IGMapOnBSTKeySortedSet`	`imapbst.h`	B* tree
`IMapOnSortedLinkedSequence`	`imapsls.h`	Linked sequence
`IGMapOnSortedLinkedSequence`	`imapsls.h`	Linked sequence
`IMapOnSortedTabularSequence`	`imapsts.h`	Tabular sequence
`IGMapOnSortedTabularSequence`	`imapsts.h`	Tabular sequence
`IMapOnSortedDilutedSequence`	`imapsds.h`	Diluted sequence
`IGMapOnSortedDilutedSequence`	`imapsds.h`	Diluted sequence
`IMapOnHashKeySet`	`imaphks.h`	Hash table
`IGMapOnHashKeySet`	`imaphks.h`	Hash table

Members

All members of flat collections are described in "Introduction to Flat Collections" on page 97. The following members are provided for map:

Method	Page	Method	Page
Constructor	101	elementAt	115
Copy Constructor	101	elementWithKey	115
Destructor	101	intersectionWith	117
operator!=	102	isBounded	117
operator=	102	isEmpty	117
operator==	102	isFull	118
add	103	key	118
addAllFrom	104	locate	118
addDifference	107	locateElementWithKey	119
addIntersection	108	locateOrAdd	121
addOrReplaceElementWithKey	109	locateOrAddElementWithKey	122
addUnion	110	maxNumberOfElements	123
allElementsDo	110	newCursor	123
anyElement	112	numberOfElements	123
contains	113	remove	125
containsAllFrom	113	removeAll	125
containsAllKeysFrom	113	removeAt	126
containsElementWithKey	113	removeElementWithKey	127
differenceWith	114	replaceAt	128

Method	Page	Method	Page
replaceElementWithKey	129	setToNext	130
setToFirst	129	unionWith	132

Map also defines a cursor that inherits from `IElementCursor`. The members for `IElementCursor` are described in "Cursor" on page 267.

Template Arguments and Required Functions

Map

```
IMap  <Element, Key>
IGMap <Element, Key, EKCOps>
```

The default implementation of the class `IMap` requires the following element and key-type functions:

Element Type

- Copy constructor
- Destructor
- Assignment
- Equality test
- Key access

Key Type

Ordering relation

Map on B* Key Sorted Set

```
IMapOnBSTKeySortedSet  <Element, Key>
IGMapOnBSTKeySortedSet <Element, Key, EKCOps>
```

The implementation of the class `IMapOnBSTKeySortedSet` requires the following element and key-type functions:

Element Type

- Default constructor
- Copy constructor
- Destructor
- Assignment
- Equality test
- Key access

Key Type

Ordering relation

Map on Sorted Linked Sequence

```
IMapOnSortedLinkedSequence  <Element, Key>
IGMapOnSortedLinkedSequence <Element, Key, EKCOps>
```

The implementation of the class `IMapOnSortedLinkedSequence` requires the following element and key-type functions:

Element Type

- Copy constructor
- Destructor
- Assignment
- Equality test
- Key access

Key Type

Ordering relation

Map on Sorted Tabular Sequence

```
IMapOnSortedTabularSequence  <Element, Key>
IGMapOnSortedTabularSequence <Element, Key, EKCOps>
```

The implementation of the class `IMapOnSortedTabularSequence` requires the following element and key-type functions:

Element Type

- Default constructor
- Copy constructor
- Destructor
- Assignment
- Equality test
- Key access

Key Type

Ordering relation

Map on Sorted Diluted Sequence

```
IMapOnSortedDilutedSequence  <Element, Key>
IGMapOnSortedDilutedSequence <Element, Key, EKCOps>
```

The implementation of the class `IMapOnSortedDilutedSequence` requires the following element and key-type functions:

Element Type

- Default constructor
- Copy constructor
- Destructor
- Assignment
- Equality test
- Key access

Key Type

Ordering relation

Map on Hash Key Set

```
IMapOnHashKeySet  <Element, Key>
IGMapOnHashKeySet <Element, Key, EKEHOps>
```

The implementation of the class `IMapOnHashKeySet` requires the following element and key-type functions:

Element Type

- Copy constructor
- Destructor
- Assignment
- Equality test
- Key access

Key Type

- Equality test
- Hash function

Abstract Class and Reference Class

For polymorphism, you can use the corresponding abstract class, IAMap, which is found in the iamap.h header file, or the corresponding reference class, IRMap, which is found in the irmap.h header file. See Chapter 11, "Polymorphic Use of Collections" in the *Open Class Library User's Guide* for further information.

Template Arguments and Required Functions

IAMap <*Element*, *Key*>
IRMap <*Element*, *Key*, *ConcreteBase*>

The concrete base class is one of the classes defined above.

The required functions are the same as the required functions of the concrete base class.

Coding Example for Map

The following program translates a string from EBCDIC to ASCII and from ASCII to EBCDIC. It uses two maps, one with the EBCDIC code as key (E2AMap) and one with the ASCII code as key (A2EMap). It converts from EBCDIC to ASCII by finding the element whose key matches the EBCDIC code in E2AMap (which has the EBCDIC code as key) and taking the ASCII code information from that element. It converts from ASCII to EBCDIC by finding the key that matches the ASCII code in A2EMap (which has the ASCII code as key) and taking the EBCDIC code information for that element.

The program uses the add() function to build the maps and the elementWithKey() function to convert the characters.

See Appendix A, "Header Files for Collection Class Library Coding Examples" on page 575 for the code of the transelm.h file.

```
// transtab.C  -  An example of using a Map
#include "transelm.h"

       // Get the standard operation classes:
#include <istdops.h>

#include "trmapops.h"

       //    char const translationTable[256] =  ....
#include "xebc2asc.h"
```

```
/*-------------------------------------------------------------*\
|  Now we define the two Map templates and two maps.            |
|  We want both of them to be based on the Hashtable KeySet.    |
\*-------------------------------------------------------------*/
#include <imaphks.h>

typedef IGMapOnHashKeySet
         < TranslationElement, char, TranslationOpsE2A >  TransE2AMap;

typedef IGMapOnHashKeySet
         < TranslationElement, char, TranslationOpsA2E >  TransA2EMap;

void display(char*, char*);

int main(int argc, char* argv[])  {

   TransA2EMap  A2EMap;
   TransE2AMap  E2AMap;

       /*-----------------------------------------------------*\
       |  Load the translation table into both maps.           |
       |  The maps organize themselves according to the key    |
       |  specification already given.                         |
       \*-----------------------------------------------------*/
   for (int i=0; i < 256; i++)
   {
                     /*      ascCode          ebcCode      */
      TranslationElement te(translationTable[i],   i   );

      E2AMap.add(te);
      A2EMap.add(te);
   }
       // What do we want to convert now?
   char* toConvert;
   if  (argc > 1)  toConvert = argv[1];
   else            toConvert = "$7  (=Dollar seven)";

   size_t textLength = strlen(toConvert) +1;

   char* convertedToAsc = new char[textLength];
   char* convertedToEbc = new char[textLength];

       // Convert the strings in place, character by character
   for (i=0; toConvert[i] != 0x00; i++)   {
      convertedToAsc[i]
        = E2AMap.elementWithKey(toConvert[i]).ascCode ();
      convertedToEbc[i]
        = A2EMap.elementWithKey(toConvert[i]).ebcCode ();
   }

   display("To convert", toConvert);
   display("After EBCDIC-ASCII conversion", convertedToAsc);
   display("After ASCII-EBCDIC conversion", convertedToEbc);

   delete[] convertedToAsc;
   delete[] convertedToEbc;

   return 0;
}
```

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

void display (char* title, char* text)  {
  cout << endl << title << ':' << endl;
  cout << "  Text: '" << text << "'" << endl;
  cout << "  Hex:   " << hex;
  for (int i=0; text[i] != 0x00; i++)    {
     cout << (int)(unsigned) text[i] << " ";
  }
  cout << dec << endl;
}
```

The program produces the following output:

```
To convert:
  Hex:   24 37 20 20 28 3d 44 6f 6c 6c 61 72 20 73 65 76 65 6e 29

After EBCDIC-ASCII conversion:
  Hex:   86 4 81 81 89 15 eb 3f 25 25 2f 94 81 b0 dd fc dd 3e 91

After ASCII-EBCDIC conversion:
  Hex:   5b f7 40 40 4d 7e c4 96 93 93 81 99 40 a2 85 a5 85 95 5d
```

Priority Queue

A *priority queue* is a key sorted bag with restricted access. It is an ordered collection of zero or more elements. Keys and multiple elements are supported. Element equality is not supported.

When an element is added, it is placed in the queue according to its key value or *priority*. The highest priority is indicated by the lowest key value. You can only remove the element with the highest priority. Within the priority queue, elements are sorted according to ascending key values, as in other key collections. You can only remove the element with the lowest key value.

For elements with equal priority, the priority queue has a first-in, first-out behavior.

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 the customer'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.

Derivation

Key Sorted Collection
  Key Sorted Bag
    Priority Queue

Note that priority queue is based on key sorted bag but is not actually derived from it or from the other classes shown above. The diagram does not show all bases of priority queue. See Figure 2 on page 4 in the *Open Class Library User's Guide* for a complete illustration. See “Restricted Access” in the *Open Class Library User's Guide* for further details.

Variants and Header Files

`IPriorityQueue`, the first class in the table below, is the default implementation variant. If you want to use polymorphism, you can replace the following class implementation variants by the reference class.

To use Visual Builder features with your collections, change the name of the desired collection class template in the list below from `I...` to `IV...`, and use the `ivprioqu.h` header file instead of the header file that you would normally use without Visual Builder.

Priority Queue

Class Name	Header File	Implementation Variant
`IPriorityQueue`	`iprioqu.h`	Linked sequence
`IGPriorityQueue`	`iprioqu.h`	Linked sequence
`IPriorityQueueOnSortedTabularSequence`	`ipqusts.h`	Tabular sequence
`IGPriorityQueueOnSortedTabularSequence`	`ipqusts.h`	Tabular sequence
`IPriorityQueueOnSortedDilutedSequence`	`ipqusds.h`	Diluted sequence
`IGPriorityQueueOnSortedDilutedSequence`	`ipqusds.h`	Diluted sequence

Members All members of flat collections are described in "Introduction to Flat Collections" on page 97. The following members are provided for priority queue:

Method	Page	Method	Page
Constructor	101	isFull	118
Copy Constructor	101	isLast	118
Destructor	101	key	118
operator=	102	lastElement	118
add	103	locateElementWithKey	119
addAllFrom	104	locateNextElementWithKey	120
allElementsDo	110	locateOrAddElementWithKey	122
anyElement	112	maxNumberOfElements	123
compare	112	newCursor	123
containsAllKeysFrom	113	numberOfDifferentKeys	123
containsElementWithKey	113	numberOfElements	123
dequeue	114	numberOfElementsWithKey	123
elementAt	115	removeAll	125
elementAtPosition	115	removeFirst	127
elementWithKey	115	setToFirst	129
enqueue	116	setToLast	129
firstElement	117	setToNext	130
isBounded	117	setToNextWithDifferentKey	130
isEmpty	117	setToPosition	131
isFirst	117	setToPrevious	131

Priority queue also defines a cursor that inherits from `IOrderedCursor`. 🕮 The members for `IOrderedCursor` are described in "Cursor" on page 267.

Template Arguments and Required Functions

Priority Queue

```
IPriorityQueue  <Element, Key>
IGPriorityQueue <Element, Key, KCOps>
```

The implementation of the class `IPriorityQueue` requires the following element and key-type functions:

Element Type

- Copy constructor
- Destructor
- Assignment
- Key access

Key Type

Ordering relation

Priority Queue on Sorted Tabular Sequence

```
IPriorityQueueOnSortedTabularSequence  <Element, Key>
IGPriorityQueueOnSortedTabularSequence <Element, Key, KCOps>
```

The implementation of the class `IPriorityQueueOnSortedTabularSequence` requires the following element and key-type functions:

Element Type

- Default constructor
- Copy constructor
- Destructor
- Assignment
- Key access

Key Type

Ordering relation

Priority Queue on Sorted Diluted Sequence

`IPriorityQueueOnSortedDilutedSequence <`*`Element`*`, `*`Key`*`>`
`IGPriorityQueueOnSortedDilutedSequence <`*`Element`*`, `*`Key`*`, `*`KCOps`*`>`

The implementation of the class `IPriorityQueueOnSortedDilutedSequence` requires the following element and key-type functions:

Element Type

- Default constructor
- Copy constructor
- Destructor
- Assignment
- Key access

Key Type

Ordering relation

Abstract Class and Reference Class

For polymorphism, you can use the corresponding abstract class, `IAPriorityQueue`, which is found in the `iaprioqu.h` header file, or the corresponding reference class, `IRPriorityQueue`, which is found in the `irprioqu.h` header file. 🕮 See Chapter 11, “Polymorphic Use of Collections” in the *Open Class Library User's Guide* for further information.

Template Arguments and Required Functions

`IAPriorityQueue <`*`Element`*`, `*`Key`*`>`
`IRPriorityQueue <`*`Element`*`, `*`Key`*`, `*`ConcreteBase`*`>`

The concrete base class is one of the classes defined above.

The required functions are the same as the required functions of the concrete base class.

Queue

A *queue* is a sequence with restricted access. It is an ordered collection of elements with no key and no element equality. The elements are arranged so that each collection has a first and a last element, each element except the last has a next element, and each element but the first has a previous element. The type and value of the elements are irrelevant, and have no effect on the behavior of the collection.

You can only add an element as the last element, and you can only remove the first element. Consequently, the elements of a queue are in chronological order.

A queue is characterized by a first-in, first-out (FIFO) behavior.

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.

Derivation

- Collection
  - Ordered Collection
    - Sequential Collection
      - Sequence
        - Queue

Note that queue is based on sequence but is not actually derived from it or from the other classes shown above. See “Restricted Access” in the *Open Class Library User's Guide* for further details.

Variants and Header Files

`IQueue`, the first class in the table below, is the default implementation variant. If you want to use polymorphism, you can replace the following class implementation variants by the reference class.

To use Visual Builder features with your collections, change the name of the desired collection class template in the list below from `I...` to `IV...`, and use the `ivqueue.h` header file instead of the header file that you would normally use without Visual Builder.

Queue

Class Name	Header File	Implementation Variant
`IQueue`	`iqueue.h`	Linked sequence
`IGQueue`	`iqueue.h`	Linked sequence
`IQueueOnTabularSequence`	`iquets.h`	Tabular sequence
`IGQueueOnTabularSequence`	`iquets.h`	Tabular sequence
`IQueueOnDilutedSequence`	`iqueds.h`	Diluted sequence
`IGQueueOnDilutedSequence`	`iqueds.h`	Diluted sequence

Members All members of flat collections are described in "Introduction to Flat Collections" on page 97. The following members are provided for queue:

Method	Page	Method	Page
Constructor	101	isBounded	117
Copy Constructor	101	isEmpty	117
Destructor	101	isFirst	117
operator=	102	isFull	118
add	103	isLast	118
addAllFrom	104	lastElement	118
addAsLast	105	maxNumberOfElements	123
allElementsDo	110	newCursor	123
anyElement	112	numberOfElements	123
compare	112	removeAll	125
dequeue	114	removeFirst	127
elementAt	115	setToFirst	129
elementAtPosition	115	setToLast	129
enqueue	116	setToNext	130
firstElement	117	setToPosition	131

Queue also defines a cursor that inherits from `IOrderedCursor`. The members for `IOrderedCursor` are described in "Cursor" on page 267.

Template Arguments and Required Functions

Queue

```
IQueue  <Element>
IGQueue <Element, StdOps>
```

The default implementation of the class `IQueue` requires the following element functions:

Element Type

- Copy constructor
- Destructor
- Assignment

Queue on Tabular Sequence

```
IQueueOnTabularSequence  <Element>
IGQueueOnTabularSequence <Element, StdOps>
```

The implementation of the class `IDequeOnTabularSequence` requires the following element functions:

Element Type

- Copy constructor
- Destructor
- Assignment

Queue on Diluted Sequence

```
IQueueOnDilutedSequence  <Element>
IGQueueOnDilutedSequence <Element, StdOps>
```

The implementation of the class `IQueueOnDilutedSequence` requires the following element functions:

Element Type

- Copy constructor
- Destructor
- Assignment

Abstract Class and Reference Class

For polymorphism, you can use the corresponding abstract class, `IAQueue`, which is found in the `iaqueue.h` header file, or the corresponding reference class, `IRQueue`, which is found in the `irqueue.h` header file. See Chapter 11, “Polymorphic Use of Collections” in the *Open Class Library User's Guide* for further information.

Template Arguments and Required Functions

```
IAQueue <Element>
IRQueue <Element, ConcreteBase>
```

The concrete base class is one of the classes defined above.

The required functions are the same as the required functions of the concrete base class.

Relation

A *relation* is an unordered collection of zero or more elements that have a key. Element equality is supported, and the values of the elements are relevant.

The keys of the elements are not unique. You can add an element whether or not there is already an element in the collection with the same key.

Figure 8 in the *Open Class Library User's Guide* illustrates the differences in behavior between map, relation, key set, and key bag when identical elements and elements with the same key are added.

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.

Figure 7 in the *Open Class Library User's Guide* gives an overview of the properties of a relation and its relationship to other flat collections.

Derivation

```
                Collection
Key Collection           Equality Collection
          Equality Key Collection
                  Relation
```

Variants and Header Files

`IRelation` is the default implementation variant. `IGRelation` is the default implementation with generic operations class. Both variants are declared in the header file `irel.h`.

To use Visual Builder features with your collections, use `IVRelation` instead of `IRelation`, and `IVGRelation` instead of `IGRelation`. Both variants are declared in the header file `ivrel.h`.

Relation

Members All members of flat collections are described in "Introduction to Flat Collections" on page 97. The following members are provided for relation:

Method	Page	Method	Page
Constructor	101	isFull	118
Copy Constructor	101	key	118
Destructor	101	locate	118
operator!=	102	locateElementWithKey	119
operator=	102	locateNextElementWithKey	120
operator==	102	locateOrAdd	121
add	103	locateOrAddElementWithKey	122
addAllFrom	104	maxNumberOfElements	123
addDifference	107	newCursor	123
addIntersection	108	numberOfDifferentKeys	123
addOrReplaceElementWithKey	109	numberOfElements	123
addUnion	110	numberOfElementsWithKey	123
allElementsDo	110	remove	125
anyElement	112	removeAll	125
contains	113	removeAllElementsWithKey	126
containsAllFrom	113	removeAt	126
containsAllKeysFrom	113	removeElementWithKey	127
containsElementWithKey	113	replaceAt	128
differenceWith	114	replaceElementWithKey	129
elementAt	115	setToFirst	129
elementWithKey	115	setToNext	130
intersectionWith	117	setToNextWithDifferentKey	130
isBounded	117	unionWith	132
isEmpty	117		

Relation also defines a cursor that inherits from `IElementCursor`. The members for `IElementCursor` are described in "Cursor" on page 267.

Template Arguments and Required Functions

```
IRelation  <Element, Key>
IGRelation <Element, Key, EKEHOps>
```

The default implementation of the class `IRelation` requires the following element functions:

Element Type

- Copy constructor
- Destructor
- Assignment

- Key access
- Equality test

Key Type

- Equality test
- Hash function

Abstract Class and Reference Class

For polymorphism, you can use the corresponding abstract class, `IARelation`, which is found in the `iarel.h` header file, or the corresponding reference class, `IRRelation`, which is found in the `irrel.h` header file. See Chapter 11, “Polymorphic Use of Collections” on page 143 in the *Open Class Library User's Guide* for further information.

Template Arguments and Required Functions

```
IARelation <Element, Key>
IRRelation <Element, Key, ConcreteBase>
```

The concrete base class is one of the classes defined above.

The required functions are the same as the required functions of the concrete base class.

Relation

Sequence

A *sequence* is an ordered collection of elements. The elements are arranged so that each collection has a first and a last element, each element except the last has a next element, and each element but the first has a previous element.

The type and value of the elements are irrelevant, and have no effect on the behavior of the collection. Elements can be added and deleted from any position in the collection. Elements can be retrieved or replaced. A sequence does not support element equality or a key. If you require element equality for a sequence, you can use an equality sequence. See “Equality Sequence” on page 145 for further details.

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.

Figure 7 in the *Open Class Library User's Guide* illustrates the properties of a sequence and its relationship to other flat collections.

Derivation

Collection
  Ordered Collection
    Sequential Collection
      Sequence

Variants and Header Files

`ISequence`, the first class in the table below, is the default implementation variant. If you want to use polymorphism, you can replace the following class implementation variants by the reference class.

To use Visual Builder features with your collections, change the name of the desired collection class template in the list below from `I...` to `IV...`, and use the `ivseq.h` header file instead of the header file that you would normally use without Visual Builder.

Sequence

Class Name	Header File	Implementation Variant
`ISequence`	`iseq.h`	Linked sequence
`IGSequence`	`iseq.h`	Linked sequence
`ILinkedSequence`	`ilnseq.h`	Linked sequence
`IGLinkedSequence`	`ilnseq.h`	Linked sequence
`ITabularSequence`	`itbseq.h`	Tabular sequence
`IGTabularSequence`	`itbseq.h`	Tabular sequence
`IDilutedSequence`	`itdseq.h`	Diluted sequence
`IGDilutedSequence`	`itdseq.h`	Diluted sequence

Members All members of flat collections are described in "Introduction to Flat Collections" on page 97. The following members are provided for sequence:

Method	Page	Method	Page
Constructor	101	isFirst	117
Copy Constructor	101	isFull	118
Destructor	101	isLast	118
operator=	102	lastElement	118
add	103	maxNumberOfElements	123
addAllFrom	104	newCursor	123
addAsFirst	105	numberOfElements	123
addAsLast	105	removeAll	125
addAsNext	106	removeAt	126
addAsPrevious	106	removeAtPosition	127
addAtPosition	107	removeFirst	127
allElementsDo	110	removeLast	128
anyElement	112	replaceAt	128
compare	112	setToFirst	129
elementAt	115	setToLast	129
elementAtPosition	115	setToNext	130
firstElement	117	setToPosition	131
isBounded	117	setToPrevious	131
isEmpty	117	sort	131

Sequence also defines a cursor that inherits from `IOrderedCursor`. 📖 The members for `IOrderedCursor` are described in "Cursor" on page 267.

Template Arguments and Required Functions

Sequence

```
ISequence  <Element>
IGSequence <Element, StdOps>
```

The default implementation of `ISequence` requires the following element functions:

Element Type

- Copy constructor
- Destructor
- Assignment

Linked Sequence

```
ILinkedSequence  <Element>
IGLinkedSequence <Element, StdOps>
```

The implementation of the class `ILinkedSequence` requires the following element functions:

Element Type

- Copy constructor
- Destructor
- Assignment

Tabular Sequence

```
ITabularSequence  <Element>
IGTabularSequence <Element, StdOps>
```

The implementation of the class `ITabularSequence` requires the following element functions:

Element Type

- Default constructor
- Copy constructor
- Destructor
- Assignment

Diluted Sequence

```
IDilutedSequence  <Element>
IGDilutedSequence <Element, StdOps>
```

The implementation of the class `IDilutedSequence` requires the following element functions:

Element Type

- Copy constructor
- Destructor
- Assignment

Abstract Class and Reference Class

For polymorphism, you can use the corresponding abstract class, `IASequence`, which is found in the `iaseq.h` header file, or the corresponding reference class, `IRSequence`, which is found in the `irseq.h` header file. See Chapter 11, "Polymorphic Use of Collections" in the *Open Class Library User's Guide* for further information.

Template Arguments and Required Functions

```
IASequence <Element>
IRSequence <Element, ConcreteBase>
```

The concrete base class is one of the sequence classes.

The required functions are the same as the required functions of the concrete base class.

Coding Example for Sequence

The following program creates a sequence using the default sequence class, `ISequence`, and adds a number of words to it. The program sorts the words in ascending order and searches the sequence for the word "fox." Finally, it prints the word that is in position 9.

The program uses two types of iteration. It uses the iterator class, `IIterator`, when printing the sequence, and it uses cursor iteration when searching for a word. With the iterator object, the program uses the `allElementsDo()` function. With cursor iteration, it uses the `setToFirst()`, `isValid()`, and `setToNext()` functions. It uses the `elementAt()` and `elementAtPosition()` functions to find words in the sequence.

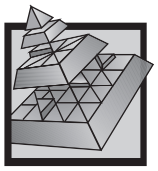
See Appendix A, "Header Files for Collection Class Library Coding Examples" on page 575 for the code of the `toyword.h` file.

```
// wordseq.C  -  An example of using a Sequence
#include <iostream.h>

              // Get definition and declaration of class Word:
#include "toyword.h"

              // Define a compare function to be used for sort:
inline long wordCompare ( Word const& w1, Word const& w2) {
   return (w1.getWord() > w2.getWord());
}

// We want to use the default Sequence called ISequence.
#include <iseq.h>

typedef ISequence <Word> WordSeq;
typedef IIterator <Word> WordIter;


// Test variables to put into the Sequence.

IString wordArray[9] = {
   "the",    "quick",   "brown",   "fox",   "jumps",
   "over",   "a",       "lazy",    "dog"
};



// Our Iterator class for use with allElementsDo().

// The alternative method of iteration, using a cursor, does
// not need such an iterator class.
class PrintClass : public WordIter
{
public:
   IBoolean applyTo(Word &w)
      {
      cout << endl << w.getWord();    // Print the string
      return(True);
      }
};




// Main program
int main()  {
   WordSeq WL;
   WordSeq::Cursor cursor(WL);
   PrintClass Print;

   int i;

   for (i = 0; i < 9; i ++) {     // Put all strings into Sequence
      Word aWord(wordArray[i]);   // Fill object with right value
      WL.addAsLast(aWord);        // Add it to the Sequence at end
   }

   cout << endl << "Sequence in initial order:" << endl;
   WL.allElementsDo(Print);

   WL.sort(wordCompare);       // Sort the Sequence ascending
   cout << endl << endl << "Sequence in sorted order:" << endl;
   WL.allElementsDo(Print);
```

```
      // Use iteration via cursor now:

      cout << endl << endl << "Look for \"fox\" in the Sequence:" << endl;
      for (cursor.setToFirst();
           cursor.isValid() && (WL.elementAt(cursor).getWord() != "fox");
           cursor.setToNext());

      if (WL.elementAt(cursor).getWord() != "fox") {
          cout << endl << "The element was not found." << endl;
      }
      else {
          cout << endl << " The element was found." << endl;
      }

      cout << endl << "The element at position 9: "
           << WL.elementAtPosition(9).getWord()
           << endl;

      return(0);
   }
```

The program produces the following output:

```
Sequence in initial order:

the
quick
brown
fox
jumps
over
a
lazy
dog

Sequence in sorted order:

a
brown
dog
fox
jumps
lazy
over
quick
the

Look for "fox" in the Sequence:

 The element was found.


The element at position 9: the
```



Set

A *set* is an unordered collection of zero or more elements with no key. Element equality is supported, and the values of the elements are relevant.

Only unique elements are supported. A request to add an element that already exists is ignored.

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.

Figure 7 in the *Open Class Library User's Guide* gives an overview of the properties of a set and its relationship to other flat collections.

The set also offers typical set functions such as union, intersection, and difference.

Derivation

Collection
  Equality Collection
    Set

Variants and Header Files

`ISet`, the first class in the table below, is the default implementation variant. If you want to use polymorphism, you can replace the following class implementation variants by the reference class.

To use Visual Builder features with your collections, change the name of the desired collection class template in the list below from `I...` to `IV...`, and use the `ivset.h` header file instead of the header file that you would normally use without Visual Builder.

Set

Class Name	Header File	Implementation Variant
`ISet`	`iset.h`	AVL tree
`IGSet`	`iset.h`	AVL tree
`ISetOnBSTKeySortedSet`	`isetbst.h`	B* tree
`IGSetOnBSTKeySortedSet`	`isetbst.h`	B* tree
`ISetOnSortedLinkedSequence`	`isetsls.h`	Linked sequence
`IGSetOnSortedLinkedSequence`	`isetsls.h`	Linked sequence
`ISetOnSortedTabularSequence`	`isetsts.h`	Tabular sequence
`IGSetOnSortedTabularSequence`	`isetsts.h`	Tabular sequence
`ISetOnSortedDilutedSequence`	`isetsds.h`	Diluted sequence
`IGSetOnSortedDilutedSequence`	`isetsds.h`	Diluted sequence
`ISetOnHashKeySet`	`isethks.h`	Hash table
`IGSetOnHashKeySet`	`isethks.h`	Hash table

Members All members of flat collections are described in “Introduction to Flat Collections” on page 97. The following members are provided for set:

Method	Page	Method	Page
Constructor	101	intersectionWith	117
Copy Constructor	101	isBounded	117
Destructor	101	isEmpty	117
operator!=	102	isFull	118
operator=	102	locate	118
operator==	102	locateOrAdd	121
add	103	maxNumberOfElements	123
addAllFrom	104	newCursor	123
addDifference	107	numberOfElements	123
addIntersection	108	remove	125
addUnion	110	removeAll	125
allElementsDo	110	removeAt	126
anyElement	112	replaceAt	128
contains	113	setToFirst	129
containsAllFrom	113	setToNext	130
differenceWith	114	unionWith	132
elementAt	115		

Set also defines a cursor that inherits from `IElementCursor`. 🕮 The members for `IElementCursor` are described in “Cursor” on page 267.

Template Arguments and Required Functions

Set

```
ISet  <Element>
IGSet <Element, ECOps>
```

The default implementation of the class `ISet` requires the following element functions:

Element Type

- Copy constructor
- Destructor
- Assignment
- Equality test
- Ordering relation

Set on B* Key Sorted Set

```
ISetOnBSTKeySortedSet  <Element>
IGSetOnBSTKeySortedSet <Element, ECOps>
```

The implementation of the class `ISetOnBSTKeySortedSet` requires the following element functions:

Element Type

- Default constructor
- Copy constructor
- Destructor
- Assignment
- Equality test
- Ordering relation

Set on Sorted Linked Sequence

```
ISetOnSortedLinkedSequence  <Element>
IGSetOnSortedLinkedSequence <Element, ECOps>
```

The implementation of the class `ISetOnSortedLinkedSequence` requires the following element functions:

Element Type

- Copy constructor
- Destructor
- Assignment
- Equality test
- Ordering relation

Set on Sorted Tabular Sequence

`ISetOnSortedTabularSequence` *<Element>*
`IGSetOnSortedTabularSequence` *<Element, ECOps>*

The implementation of the class `ISetOnSortedTabularSequence` requires the following element functions:

Element Type

- Default constructor
- Copy constructor
- Destructor
- Assignment
- Equality test
- Ordering relation

Set on Sorted Diluted Sequence

`ISetOnSortedDilutedSequence` *<Element>*
`IGSetOnSortedDilutedSequence` *<Element, ECOps>*

The implementation of the class `ISetOnSortedDilutedSequence` requires the following element functions:

Element Type

- Default constructor
- Copy constructor
- Destructor
- Assignment
- Equality test
- Ordering relation

Set on Hash Key Set

`ISetOnHashKeySet` *<Element>*
`IGSetOnHashKeySet` *<Element, EHOps>*

The implementation of the class `ISetOnHashKeySet` requires the following element functions:

Element Type

- Copy constructor
- Destructor
- Assignment
- Equality test
- Hash function

Abstract Class and Reference Class

For polymorphism, you can use the corresponding abstract class, `IASet`, which is found in the `iaset.h` header file, or the corresponding reference class, `IRSet`, which is found in the `irset.h` header file. See Chapter 11, "Polymorphic Use of Collections" in the *Open Class Library User's Guide* for further information.

Template Arguments and Required Functions

```
IASet <Element>
IRSet <Element, ConcreteBase>
```

The concrete base class is one of the classes defined above.

The required functions are the same as the required functions of the concrete base class.

Coding Example for Set

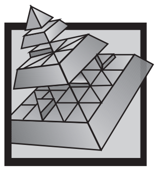
The follow program creates sets using the default class, `ISet`. The `odd` set contains all odd numbers less than ten. The `prime` set contains all prime numbers less than ten. The program creates a set, `oddPrime`, that contains all the prime numbers less than ten that are odd, by using the intersection of `odd` and `prime`. It creates another set, `evenPrime`, that contains all the prime numbers less than ten that are even, by using the difference of `prime` and `oddPrime`.

When printing the sets, the program uses the iterator class, `IIterator`. It uses the `add()` function to build the `odd` and `prime` sets. It uses the `addIntersection()` and `addDifference()` functions to create the `oddPrime` and `evenPrime` sets, respectively.

```
// evenodd.C  -  An example of using a Set
#include <iostream.h>

#include <iset.h>        // Take the defaults for the Set and for
                         // the required functions for integer
typedef ISet <int> IntSet;

// For iteration we want to use an object of an iterator class
class PrintClass : public IIterator<int>  {
  public:
    virtual IBoolean applyTo(int& i)
      { cout << " " << i << " "; return True;}
};

// Local prototype for the function to display an IntSet.
void    List(char *, IntSet &);

// Main program
int main ()  {
   IntSet odd, prime;
   IntSet oddPrime, evenPrime;

   int One = 1, Two = 2, Three = 3, Five = 5, Seven = 7, Nine = 9;
// Fill odd set with odd integers < 10
   odd.add( One );
   odd.add( Three );
   odd.add( Five );
   odd.add( Seven );
   odd.add( Nine );
   List("Odds less than 10:  ", odd);

// Fill prime set with primes < 10
   prime.add( Two );
   prime.add( Three );
   prime.add( Five );
   prime.add( Seven );
   List("Primes less than 10:  ", prime);

// Intersect 'Odd' and 'Prime' to give 'OddPrime'
   oddPrime.addIntersection( odd, prime);
   List("Odd primes less than 10:  ", oddPrime);

// Subtract all 'Odd' from 'Prime' to give 'EvenPrime'
   evenPrime.addDifference( prime, oddPrime);
   List("Even primes less than 10:  ", evenPrime);

   return(0);
}

// Local function to display an IntSet.

void List(char *Message, IntSet &anIntSet)  {
   PrintClass Print;

   cout << Message;
   anIntSet.allElementsDo(Print);
   cout << endl;
}
```

The program produces the following output:

```
Odds less than 10:   1  3  5  7  9
Primes less than 10:   2  3  5  7
Odd primes less than 10:   3  5  7
Even primes less than 10:   2
```



Sorted Bag

A *sorted bag* is an ordered collection of zero or more elements with no key. Both element equality and multiple elements are supported.

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.

Figure 7 in the *Open Class Library User's Guide* gives an overview of the properties of a sorted bag and its relationship to other flat collections.

Derivation

```
           Collection
                      Ordered Collection
Equality Collection     Sorted Collection
        Equality Sorted Collection
              Sorted Bag
```

Variants and Header Files

`ISortedBag`, the first class in the table below, is the default implementation variant. If you want to use polymorphism, you can replace the following class implementation variants by the reference class.

To use Visual Builder features with your collections, change the name of the desired collection class template in the list below from `I...` to `IV...`, and use the `ivsrtbag.h` header file instead of the header file that you would normally use without Visual Builder.

Class Name	Header File	Implementation Variant
`ISortedBag`	`isrtbag.h`	AVL tree
`IGSortedBag`	`isrtbag.h`	AVL tree
`ISortedBagOnBSTKeySortedSet`	`isbbst.h`	B* tree
`IGSortedBagOnBSTKeySortedSet`	`isbbst.h`	B* tree
`ISortedBagOnSortedLinkedSequence`	`isbsls.h`	Linked sequence
`IGSortedBagOnSortedLinkedSequence`	`isbsls.h`	Linked sequence

Sorted Bag

Class Name	Header File	Implementation Variant
`ISortedBagOnSortedTabularSequence`	`isbsts.h`	Tabular sequence
`IGSortedBagOnSortedTabularSequence`	`isbsts.h`	Tabular sequence
`ISortedBagOnSortedDilutedSequence`	`isbsds.h`	Diluted sequence
`IGSortedBagOnSortedDilutedSequence`	`isbsds.h`	Diluted sequence

Members

All members of flat collections are described in "Introduction to Flat Collections" on page 97. The following members are provided for sorted bag:

Method	Page	Method	Page
Constructor	101	isLast	118
Copy Constructor	101	lastElement	118
Destructor	101	locate	118
operator!=	102	locateNext	120
operator=	102	locateOrAdd	121
operator==	102	maxNumberOfElements	123
add	103	newCursor	123
addAllFrom	104	numberOfDifferentElements	123
addDifference	107	numberOfElements	123
addIntersection	108	numberOfOccurrences	124
addUnion	110	remove	125
allElementsDo	110	removeAll	125
anyElement	112	removeAllOccurrences	126
compare	112	removeAt	126
contains	113	removeAtPosition	127
containsAllFrom	113	removeFirst	127
differenceWith	114	removeLast	128
elementAt	115	replaceAt	128
elementAtPosition	115	setToFirst	129
firstElement	117	setToLast	129
intersectionWith	117	setToNext	130
isBounded	117	setToNextDifferentElement	130
isEmpty	117	setToPosition	131
isFirst	117	setToPrevious	131
isFull	118	unionWith	132

Sorted Bag also defines a cursor that inherits from `IOrderedCursor`. The members for `IOrderedCursor` are described in "Cursor" on page 267.

Template Arguments and Required Functions

Sorted Bag

`ISortedBag` *<Element>*
`IGSortedBag` *<Element, ECOps>*

The default implementation of the class `ISortedBag` requires the following element functions:

Element Type

- Default constructor
- Copy constructor
- Destructor
- Assignment
- Equality test
- Ordering relation

Sorted Bag on B* Key Sorted Set

`ISortedBagOnBSTKeySortedSet` *<Element>*
`IGSortedBagOnBSTKeySortedSet` *<Element, ECOps>*

The implementation of the class `ISortedBagOnBSTKeySortedSet` requires the following element functions:

Element Type

- Default constructor
- Copy constructor
- Destructor
- Assignment
- Equality test
- Ordering relation

Sorted Bag on Sorted Linked Sequence

`ISortedBagOnSortedLinkedSequence` *<Element>*
`IGSortedBagOnSortedLinkedSequence` *<Element, ECOps>*

The implementation of the class `ISortedBagOnSortedLinkedSequence` requires the following element functions:

Element Type

- Default constructor
- Constructor
- Assignment

- Equality test
- Ordering relation

Sorted Bag on Sorted Tabular Sequence

`ISortedBagOnSortedTabularSequence` *<Element>*
`IGSortedBagOnSortedTabularSequence` *<Element*, *ECOps>*

The implementation of the class `ISortedBagOnSortedTabularSequence` requires the following element functions:

Element Type

- Default constructor
- Copy constructor
- Destructor
- Assignment
- Equality test
- Ordering relation

Sorted Bag on Sorted Diluted Sequence

`ISortedBagOnSortedDilutedSequence` *<Element>*
`IGSortedBagOnSortedDilutedSequence` *<Element*, *ECOps>*

The implementation of the class `ISortedBagOnSortedDilutedSequence` requires the following element functions:

Element Type

- Default constructor
- Copy constructor
- Destructor
- Assignment
- Equality test
- Ordering relation

Abstract Class and Reference Class

For polymorphism, you can use the corresponding abstract class, `IASortedBag`, which is found in the `iasrtbag.h` header file, or the corresponding reference class, `IRSortedBag`, which is found in the `irsrtbag.h` header file. See Chapter 11, “Polymorphic Use of Collections” in the *Open Class Library User's Guide* for further information.

Template Arguments and Required Functions

```
IASortedBag <Element>
IRSortedBag <Element, ConcreteBase>
```

The concrete base class is one of the classes defined above.

The required functions are the same as the required functions of the concrete base class.

Sorted Bag

Sorted Map

A *sorted map* is an ordered collection of zero or more elements that have a key. Element equality is supported and the values of the elements are relevant. Elements are sorted by the value of their keys.

Only elements with unique keys are supported. A request to add an element whose key already exists in another element of the collection causes an exception to be thrown. A request to add a duplicate element is ignored.

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.

Figure 7 in the *Open Class Library User's Guide* gives an overview of the properties of a sorted map and its relationship to other flat collections.

Derivation

Equality Key Collection   Equality Sorted Collection
Equality Key Sorted Collection
Sorted Map

The diagram does not show all bases of sorted map. See Figure 2 in the *Open Class Library User's Guide* for a complete illustration.

Variants and Header Files

`ISortedMap`, the first class in the table below, is the default implementation variant. If you want to use polymorphism, you can replace the following class implementation variants by the reference class.

To use Visual Builder features with your collections, change the name of the desired collection class template in the list below from `I...` to `IV...`, and use the `ivsrtmap.h` header file instead of the header file that you would normally use without Visual Builder.

Class Name	Header File	Implementation Variant
`ISortedMap`	`isrtmap.h`	AVL tree
`IGSortedMap`	`isrtmap.h`	AVL tree
`ISortedMapOnBSTKeySortedSet`	`ismbst.h`	B* tree
`IGSortedMapOnBSTKeySortedSet`	`ismbst.h`	B* tree

Sorted Map

Class Name	Header File	Implementation Variant
`ISortedMapOnSortedLinkedSequence`	`ismsls.h`	Linked sequence
`IGSortedMapOnSortedLinkedSequence`	`ismsls.h`	Linked sequence
`ISortedMapOnSortedTabularSequence`	`ismsts.h`	Tabular sequence
`IGSortedMapOnSortedTabularSequence`	`ismsts.h`	Tabular sequence
`ISortedMapOnSortedDilutedSequence`	`ismsds.h`	Diluted sequence
`IGSortedMapOnSortedDilutedSequence`	`ismsds.h`	Diluted sequence

Members

All members of flat collections are described in "Introduction to Flat Collections" on page 97. The following members are provided for sorted maps:

Method	Page	Method	Page
Constructor	101	isFull	118
Copy Constructor	101	isLast	118
Destructor	101	key	118
operator!=	102	lastElement	118
operator=	102	locate	118
operator==	102	locateElementWithKey	119
add	103	locateNext	120
addAllFrom	104	locateNextElementWithKey	120
addDifference	107	locateOrAdd	121
addIntersection	108	locateOrAddElementWithKey	122
addOrReplaceElementWithKey	109	maxNumberOfElements	123
addUnion	110	newCursor	123
allElementsDo	110	numberOfElements	123
anyElement	112	remove	125
compare	112	removeAll	125
contains	113	removeAt	126
containsAllFrom	113	removeAtPosition	127
containsAllKeysFrom	113	removeElementWithKey	127
containsElementWithKey	113	removeFirst	127
differenceWith	114	removeLast	128
elementAt	115	replaceAt	128
elementAtPosition	115	replaceElementWithKey	129
elementWithKey	115	setToFirst	129
firstElement	117	setToLast	129
intersectionWith	117	setToNext	130
isBounded	117	setToPosition	131
isEmpty	117	setToPrevious	131
isFirst	117	unionWith	132

Sorted map also defines a cursor that inherits from `IOrderedCursor`. The members for `IOrderedCursor` are described in “Cursor” on page 267.

Template Arguments and Required Functions

Sorted Map

```
ISortedMap  <Element, Key>
IGSortedMap <Element, Key, EKCOps>
```

The implementation of the class `ISortedMap` requires the following element and key-type functions:

Element Type

- Copy constructor
- Destructor
- Assignment
- Key access
- Equality test

Key Type

Ordering relation

Sorted Map on B* Key Sorted Set

```
ISortedMapOnBSTKeySortedSet  <Element, Key>
IGSortedMapOnBSTKeySortedSet <Element, Key, EKCOps>
```

The implementation of the class `ISortedMapOnBSTKeySortedSet` requires the following element and key-type functions:

Element Type

- Default constructor
- Copy constructor
- Destructor
- Assignment
- Key access
- Equality test

Key Type

Ordering relation

Sorted Map

Sorted Map on Sorted Linked Sequence

`ISortedMapOnSortedLinkedSequence` <*Element*, *Key*>
`IGSortedMapOnSortedLinkedSequence` <*Element*, *Key*, *EKCOps*>

The implementation of the class `ISortedMapOnSortedLinkedSequence` requires the following element and key-type functions:

Element Type

- Copy constructor
- Destructor
- Assignment
- Key access
- Equality test

Key Type

Ordering relation

Sorted Map on Sorted Tabular Sequence

`ISortedMapOnSortedTabularSequence` <*Element*, *Key*>
`IGSortedMapOnSortedTabularSequence` <*Element*, *Key*, *EKCOps*>

The implementation of the class `ISortedMapOnSortedTabularSequence` requires the following element and key-type functions:

Element Type

- Default constructor
- Copy constructor
- Destructor
- Assignment
- Key access
- Equality test

Key Type

Ordering relation

Sorted Map on Sorted Diluted Sequence

`ISortedMapOnSortedDilutedSequence` <*Element*, *Key*>
`IGSortedMapOnSortedDilutedSequence` <*Element*, *Key*, *EKCOps*>

The implementation of the class `ISortedMapOnSortedDilutedSequence` requires the following element and key-type functions:

Element Type

- Default constructor
- Copy constructor
- Destructor
- Assignment
- Key access
- Equality test

Key Type

Ordering relation

Abstract Class and Reference Class

For polymorphism, you can use the corresponding abstract class, `IASortedMap`, which is found in the `iasrtmap.h` header file, or the corresponding reference class, `IRSortedMap`, which is found in the `irsrtmap.h` header file. See Chapter 11, "Polymorphic Use of Collections" in the *Open Class Library User's Guide* for further information.

Template Arguments and Required Functions

```
IASortedMap <Element, Key>
IRSortedMap <Element, Key, ConcreteBase>
```

The concrete base class is one of the classes defined above.

The required functions are the same as the required functions of the concrete base class.

Coding Example for Sorted Map

The following program uses a sorted map and a sorted relation to display sorted lists of the name and size of files contained on a disk. It uses the default classes, `ISortedMap` and `ISortedRelation`, to implement the collections. The program uses the sorted map to store the name of the file, because all elements in a sorted map are unique and all names on a disk are unique. It uses a sorted relation for the file size, because there may be identical file sizes, and identical values are permissible in sorted relations.

The program uses the `add()` function to fill both collections. To print the collections, it uses the `forCursor` macro and the `allElementsDo()` function.

Sorted Map

The program produces a list of files sorted by name (in ascending order) and a list of the same files sorted by file size (in descending order). Because the output varies depending on the file system in use when it is run, no output is shown here.

See Appendix A, “Header Files for Collection Class Library Coding Examples” on page 575 for the code of the `dsur.h` file.

```
// dskusage.C -  An example of using a Sorted Map and a Sorted Relation
#include "dsur.h"
                     // Our own common exit for all errors:
void errorExit(int, char*, char* = "");

                     // Use the default Sorted Map as is:
#include <isrtmap.h>
                     // Use the default Sorted Relation as is:
#include <isrtrel.h>

int main (int argc, char* argv[]) {
    char* fspec = "dsu.dat"; // Default for input file
    if (argc > 1)   fspec = argv[1];
    ifstream  inputfile(fspec);
    if (!inputfile)
        errorExit(20, "Unable to open input file", fspec);

    ISortedMap     <DiskSpaceUR, char*> dsurByName;
    ISortedMap     <DiskSpaceUR, char*>::Cursor
                                     curByName(dsurByName);

    IGSortedRelation <DiskSpaceUR, int, DSURBySpaceOps>
       dsurBySpace;
    IGSortedRelation <DiskSpaceUR, int, DSURBySpaceOps>::Cursor
       curBySpace(dsurBySpace);

                         // Read all records into dsurByName
    while (inputfile.good()) {
       DiskSpaceUR dsur(inputfile);
       if (dsur.isValid()) {
          dsurByName.add(dsur);
          dsurBySpace.add(dsur);
       }
    }
    if (! inputfile.eof())
            errorExit(39, "Error during read of", fspec);

    cout << "\n\nAll Disk Space Usage records "
         << "sorted (ascending) by name:\n" << endl;

    forCursor(curByName)
       cout << "  " << dsurByName.elementAt(curByName) << endl;

    cout << "\n\nAll Disk Space Usage records "
         << "sorted (descending) by space:\n" << endl;

    forCursor(curBySpace)
       cout << "  " << dsurBySpace.elementAt(curBySpace) << endl;
    return 0;
}

#include <stdlib.h>
                                   // for exit() definition
void errorExit (int rc, char* s1, char* s2) {
    cerr << s1 << " " << s2 << endl;
    exit(rc);
}
```

Sorted Relation

A *sorted relation* is an ordered collection of zero or more elements that have a key. The elements are sorted by the value of their key. Element equality is supported, and the values of the elements are relevant.

The keys of the elements are not unique. You can add an element whether or not there is already an element in the collection with the same key.

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.

Figure 7 in the *Open Class Library User's Guide* gives an overview of the properties of a sorted relation and its relationship to other flat collections.

Derivation

```
Equality Key Collection       Equality Collection
          Equality Key Sorted Collection
                  Sorted Relation
```

The diagram does not show all bases of sorted relation. See Figure 2 on page 4 in the *Open Class Library User's Guide* for a complete illustration.

Variants and Header Files

`ISortedRelation`, the first class in the table below, is the default implementation variant. If you want to use polymorphism, you can replace the following class implementation variants by the reference class.

To use Visual Builder features with your collections, change the name of the desired collection class template in the list below from `I...` to `IV...`, and use the `ivsrtrel.h` header file instead of the header file that you would normally use without Visual Builder.

Sorted Relation

Class Name	Header File	Implementation Variant
`ISortedRelation`	`isrtrel.h`	Linked sequence
`IGSortedRelation`	`isrtrel.h`	Linked sequence
`ISortedRelationOnSortedTabularSequence`	`isrsts.h`	Tabular sequence
`IGSortedRelationOnSortedTabularSequence`	`isrsts.h`	Tabular sequence
`ISortedRelationOnSortedDilutedSequence`	`isrsds.h`	Diluted sequence
`IGSortedRelationOnSortedDilutedSequence`	`isrsds.h`	Diluted sequence

Members All members of flat collections are described in "Introduction to Flat Collections" on page 97. The following members are provided for sorted relation:

Method	Page	Method	Page
Constructor	101	key	118
Copy Constructor	101	lastElement	118
Destructor	101	locate	118
operator!=	102	locateElementWithKey	119
operator=	102	locateNext	120
operator==	102	locateNextElementWithKey	120
add	103	locateOrAdd	121
addAllFrom	104	locateOrAddElementWithKey	122
addDifference	107	maxNumberOfElements	123
addIntersection	108	newCursor	123
addOrReplaceElementWithKey	109	numberOfDifferentKeys	123
addUnion	110	numberOfElements	123
allElementsDo	110	numberOfElementsWithKey	123
anyElement	112	remove	125
compare	112	removeAll	125
contains	113	removeAllElementsWithKey	126
containsAllFrom	113	removeAt	126
containsAllKeysFrom	113	removeAtPosition	127
containsElementWithKey	113	removeElementWithKey	127
differenceWith	114	removeFirst	127
elementAt	115	removeLast	128
elementAtPosition	115	replaceAt	128
elementWithKey	115	replaceElementWithKey	129
firstElement	117	setToFirst	129
intersectionWith	117	setToLast	129
isBounded	117	setToNext	130
isEmpty	117	setToNextWithDifferentKey	130
isFirst	117	setToPosition	131
isFull	118	setToPrevious	131
isLast	118	unionWith	132

Sorted relation also defines a cursor that inherits from `IOrderedCursor`. The members for `IOrderedCursor` are described in “Cursor” on page 267.

Template Arguments and Required Functions

Sorted Relation

```
ISortedRelation  <Element, Key>
IGSortedRelation <Element, Key, EKCOps>
```

The default implementation of the class `ISortedRelation` requires the following element and key-type functions:

Element Type

- Copy constructor
- Destructor
- Assignment
- Key access
- Equality test

Key Type

Ordering relation

Sorted Relation on Sorted Tabular Sequence

```
ISortedRelationOnSortedTabularSequence  <Element, Key>
IGSortedRelationOnSortedTabularSequence <Element, Key, EKCOps>
```

The implementation of the class `ISortedRelationOnSortedTabularSequence` requires the following element and key-type functions:

Element Type

- Default constructor
- Copy constructor
- Destructor
- Assignment
- Key access
- Equality test

Key Type

Ordering relation

Sorted Relation

Sorted Relation on Sorted Diluted Sequence

`ISortedRelationOnSortedDilutedSequence` *<Element, Key>*
`IGSortedRelationOnSortedDilutedSequence` *<Element, Key, EKCOps>*

The implementation of the class `ISortedRelationOnSortedDilutedSequence` requires the following element and key-type functions:

Element Type

- Default constructor
- Copy constructor
- Destructor
- Assignment
- Key access
- Equality test

Key Type

Ordering relation

Abstract Class and Reference Class

For polymorphism, you can use the corresponding abstract class, `IASortedRelation`, which is found in the `iasrtrel.h` header file, or the corresponding reference class, `IRSortedRelation`, which is found in the `irsrtrel.h` header file. 🕮 See Chapter 11, "Polymorphic Use of Collections" in the *Open Class Library User's Guide* for further information.

Template Arguments and Required Functions

`IASortedRelation` *<Element, Key>*
`IRSortedRelation` *<Element, Key,ConcreteBase>*

The concrete base class is one of the classes defined above.

The required functions are the same as the required functions of the concrete base class.

Coding Example for Sorted Relation

🕮 See "Coding Example for Sorted Map" on page 223 for an example of a sorted relation.

Sorted Set

A *sorted set* is an ordered collection of zero or more elements with element equality but no key. Only unique elements are supported. A request to add an element that already exists is ignored. The value of the elements is relevant.

The elements of a sorted set are ordered such that the value of each element is less than or equal to the value of its successor.

The element with the smallest value currently in a sorted set is called the *first* element. The element with the largest value is called the *last* element. When an element is added, it is placed in the sorted set according to the defined ordering relation.

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.

Figure 7 in the *Open Class Library User's Guide* gives an overview of the properties of a sorted set and its relationship to other flat collections.

Derivation

```
              Collection
Ordered Collection
    Sorted Collection         Equality Collection
            Equality Sorted Collection
                   Sorted Set
```

Variants and Header Files

`ISortedSet`, the first class in the table below, is the default implementation variant. If you want to use polymorphism, you can replace the following class implementation variants by the reference class.

To use Visual Builder features with your collections, change the name of the desired collection class template in the list below from `I...` to `IV...`, and use the `ivsrtset.h` header file instead of the header file that you would normally use without Visual Builder.

Class Name	Header File	Implementation Variant
`ISortedSet`	`isrtset.h`	AVL tree
`IGSortedSet`	`isrtset.h`	AVL tree
`ISortedSetOnBSTKeySortedSet`	`issbst.h`	B* tree
`IGSortedSetOnBSTKeySortedSet`	`issbst.h`	B* tree
`ISortedSetOnSortedLinkedSequence`	`isssls.h`	Linked sequence
`IGSortedSetOnSortedLinkedSequence`	`isssls.h`	Linked sequence
`ISortedSetOnSortedTabularSequence`	`isssts.h`	Tabular sequence
`IGSortedSetOnSortedTabularSequence`	`isssts.h`	Tabular sequence
`ISortedSetOnSortedDilutedSequence`	`isssds.h`	Diluted sequence
`IGSortedSetOnSortedDilutedSequence`	`isssds.h`	Diluted sequence

Members

All members of flat collections are described in “Introduction to Flat Collections” on page 97. The following members are provided for sorted sets:

Method	Page	Method	Page
Constructor	101	isFirst	117
Copy Constructor	101	isFull	118
Destructor	101	isLast	118
operator!=	102	lastElement	118
operator=	102	locate	118
operator==	102	locateNext	120
add	103	locateOrAdd	121
addAllFrom	104	maxNumberOfElements	123
addDifference	107	newCursor	123
addIntersection	108	remove	125
addUnion	110	removeAll	125
allElementsDo	110	removeAt	126
anyElement	112	removeAtPosition	127
compare	112	removeFirst	127
contains	113	removeLast	128
containsAllFrom	113	replaceAt	128
differenceWith	114	setToFirst	129
elementAt	115	setToLast	129
elementAtPosition	115	setToNext	130
firstElement	117	setToPosition	131
intersectionWith	117	setToPrevious	131
isBounded	117	unionWith	132
isEmpty	117		

Sorted Set also defines a cursor that inherits from `IOrderedCursor`. The members for `IOrderedCursor` are described in “Cursor” on page 267.

Template Arguments and Required Functions

Sorted Set

```
ISortedSet  <Element>
IGSortedSet <Element, ECOps>
```

The default implementation of the class `ISortedSet` requires the following element functions:

Element Type

- Copy constructor
- Destructor
- Assignment
- Equality test
- Ordering relation

Sorted Set on B* Key Sorted Set

```
ISortedSetOnBSTKeySortedSet  <Element>
IGSortedSetOnBSTKeySortedSet <Element, ECOps>
```

The default implementation of the class `ISortedSetOnBSTKeySortedSet` requires the following element functions:

Element Type

- Default constructor
- Copy constructor
- Destructor
- Assignment
- Equality test
- Ordering relation

Sorted Set on Sorted Linked Sequence

`ISortedSetOnSortedLinkedSequence <`*`Element`*`>`
`IGSortedSetOnSortedLinkedSequence <`*`Element`*`, `*`ECOps`*`>`

The implementation of the class `ISortedSetOnSortedLinkedSequence` requires the following element functions:

Element Type

- Copy constructor
- Assignment
- Destructor
- Equality test
- Ordering relation

Sorted Set on Sorted Tabular Sequence

`ISortedSetOnSortedTabularSequence <`*`Element`*`>`
`IGSortedSetOnSortedTabularSequence <`*`Element`*`, `*`ECOps`*`>`

The implementation of the class `ISortedSetOnSortedTabularSequence` requires the following element functions:

Element Type

- Default constructor
- Copy constructor
- Destructor
- Assignment
- Equality test
- Ordering relation

Sorted Set on Sorted Diluted Sequence

`ISortedSetOnSortedDilutedSequence <`*`Element`*`>`
`IGSortedSetOnSortedDilutedSequence <`*`Element`*`, `*`ECOps`*`>`

The implementation of the class `ISortedSetOnSortedDilutedSequence` requires the following element functions:

Element Type

- Default constructor
- Copy constructor
- Destructor
- Assignment
- Equality test
- Ordering relation

Abstract Class and Reference Class

For polymorphism, you can use the corresponding abstract class, `IASortedSet`, which is found in the `iasrtset.h` header file, or the corresponding reference class, `IRSortedSet`, which is found in the `irsrtset.h` header file. See Chapter 11, "Polymorphic Use of Collections" in the *Open Class Library User's Guide* for further information.

Template Arguments and Required Functions

```
IASortedSet <Element>
IRSortedSet <Element, ConcreteBase>
```

The concrete base class is one of the classes defined above.

The required functions are the same as the required functions of the concrete base class.

Coding Example for Sorted Set

The following program uses the default class, `ISortedSet`, to create sorted lists of planets with different properties. The program stores all planets in our solar system, all heavy planets in our solar system, all bright planets in our solar system, and all heavy or bright planets in our solar system in a number of sorted sets. Each set sorts the planets by its distance from the sun.

The program uses the `forCursor` macro to create the `heavyPlanets` and the `brightPlanets` collections. It uses the `allElementsDo()` function to display the planets in each collection and the `unionWith()` function when creating the bright-or-heavy planets category.

See Appendix A, "Header Files for Collection Class Library Coding Examples" on page 575 for the code of the `planet.h` file.

Sorted Set

```
// planets.C  -  An example of using a Sorted Set
#include <iostream.h>

                  // Let's use the Sorted Set Default Variant:
#include <isrtset.h>

                  // Get Class Planet:
#include "planet.h"


int main()    {
   ISortedSet<Planet>  allPlanets, heavyPlanets, brightPlanets;
                       // A cursor to cursor through allPlanets:
   ISortedSet<Planet>::Cursor aPCursor(allPlanets);

   SayPlanetName showPlanet;

   allPlanets.add( Planet("Earth",   149.60f,   1.0000f, 99.9f));
   allPlanets.add( Planet("Jupiter", 778.3f,  317.818f,  -2.4f));
   allPlanets.add( Planet("Mars",    227.9f,    0.1078f, -1.9f));
   allPlanets.add( Planet("Mercury",  57.91f,   0.0558f, -0.2f));
   allPlanets.add( Planet("Neptun", 4498.f,    17.216f,  +7.6f));
   allPlanets.add( Planet("Pluto",  5910.f,     0.18f,  +14.7f));
   allPlanets.add( Planet("Saturn", 1428.f,    95.112f,  +0.8f));
   allPlanets.add( Planet("Uranus", 2872.f,    14.517f,  +5.8f));
   allPlanets.add( Planet("Venus",   108.21f,   0.8148f, -4.1f));

   forCursor(aPCursor)    {
      if (allPlanets.elementAt(aPCursor).isHeavy())
         heavyPlanets.add(allPlanets.elementAt(aPCursor));

      if (allPlanets.elementAt(aPCursor).isBright())
         brightPlanets.add(allPlanets.elementAt(aPCursor));
   }

   cout << endl << endl << "All Planets: " << endl;
   allPlanets.allElementsDo(showPlanet);

   cout << endl << endl << "Heavy Planets: " << endl;
   heavyPlanets.allElementsDo(showPlanet);

   cout << endl << endl << "Bright Planets: " << endl;
   brightPlanets.allElementsDo(showPlanet);

   cout << endl << endl << "Bright-or-Heavy Planets: " << endl;
   brightPlanets.unionWith(heavyPlanets);
   brightPlanets.allElementsDo(showPlanet);

   cout << endl << endl
        << "Did you notice that all these Sets are sorted"
        << " in the same order"
        << endl
        << " (distance of planet from sun) ? " << endl;

   return 0;

}
```

The program produces the following output:

```
All Planets:
 Mercury  Venus  Earth  Mars  Jupiter  Saturn  Uranus  Neptune  Pluto

Heavy Planets:
 Jupiter  Saturn  Uranus  Neptune

Bright Planets:
 Mercury  Venus  Mars  Jupiter

Bright-or-Heavy Planets:
 Mercury  Venus  Mars  Jupiter  Saturn  Uranus  Neptune

Did you notice that all these Sets are sorted in the same order
 (distance of planet from sun) ?
```

Stack

A *stack* is a sequence with restricted access. It is an ordered collection of elements with no key and no element equality. The elements are arranged so that each collection has a first and a last element, each element except the last has a next element, and each element but the first has a previous element. The type and value of the elements are irrelevant and have no effect on the behavior of the stack.

Elements are added to and deleted from the *top* of the stack. Consequently, the elements of a stack are in reverse chronological order.

A stack is characterized by a last-in, first-out (LIFO) behavior.

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 and you can go home.

Derivation

Collection
  Ordered Collection
    Sequential Collection
      Sequence
        Stack

Note that stack is based on sequence but is not actually derived from it or from the other classes shown above. See “Restricted Access” in the *Open Class Library User's Guide* for further details.

Variants and Header Files

`IStack`, the first class in the table below, is the default implementation variant. If you want to use polymorphism you can replace the following class implementation variants by the reference class.

To use Visual Builder features with your collections, change the name of the desired collection class template in the list below from `I...` to `IV...`, and use the `ivstack.h` header file instead of the header file that you would normally use without Visual Builder.

Stack

Class Name	Header File	Implementation Variant
`IStack`	`istack.h`	Linked sequence
`IGStack`	`istack.h`	Linked sequence
`IStackOnTabularSequence`	`istkts.h`	Tabular sequence
`IGStackOnTabularSequence`	`istkts.h`	Tabular sequence
`IStackOnDilutedSequence`	`istkds.h`	Diluted sequence
`IGStackOnDilutedSequence`	`istkds.h`	Diluted sequence

Members All members of flat collections are described in "Introduction to Flat Collections" on page 97. The following members are provided for stack:

Method	Page	Method	Page
Constructor	101	isFull	118
Copy Constructor	101	isLast	118
Destructor	101	lastElement	118
operator=	102	maxNumberOfElements	123
add	103	newCursor	123
addAllFrom	104	numberOfElements	123
addAsLast	105	pop	124
allElementsDo	110	push	124
anyElement	112	removeAll	125
compare	112	removeLast	128
elementAt	115	setToFirst	129
elementAtPosition	115	setToLast	129
firstElement	117	setToNext	130
isBounded	117	setToPosition	131
isEmpty	117	setToPrevious	131
isFirst	117	top	132

Stack also defines a cursor that inherits from `IOrderedCursor`. The members for `IOrderedCursor` are described in "Cursor" on page 267.

Template Arguments and Required Functions

Stack

```
IStack  <Element>
IGStack <Element, StdOps>
```

The default implementation of the class `IStack` requires the following element functions:

Element Type

- Copy constructor
- Destructor
- Assignment

Stack on Tabular Sequence

```
IStackOnTabularSequence  <Element>
IGStackOnTabularSequence <Element, StdOps>
```

The implementation of the class `IStackOnTabularSequence` requires the following element functions:

Element Type

- Default constructor
- Copy constructor
- Destructor
- Assignment

Stack on Diluted Sequence

```
IStackOnDilutedSequence  <Element>
IGStackOnDilutedSequence <Element, StdOps>
```

The implementation of the class `IStackOnDilutedSequence` requires the following element functions:

Element Type

- Default constructor
- Copy constructor
- Destructor
- Assignment

Abstract Class and Reference Class

For polymorphism, you can use the corresponding abstract class, `IAStack`, which is found in the `iastack.h` header file, or the corresponding reference class, `IRStack`, which is found in the `irstack.h` header file. See Chapter 11, "Polymorphic Use of Collections" in the *Open Class Library User's Guide* for further information.

Template Arguments and Required Functions

```
IRStack <Element, ConcreteBase>
IAStack <Element>
```

The concrete base class is one of the classes defined above.

The required functions are the same as the required functions of the concrete base class.

Coding Example for Stack

The following program creates two stacks (Stack1 and Stack2) using the default class, IStack. It adds a number of words to Stack1, removes them from Stack1, adds them to Stack2, and finally removes them from Stack2 so that they can be printed. The push() and pop() functions are used for adding and removing elements, respectively.

Between these stack operations the stacks are printed. To prevent the stack from changing during printing, the program uses the constant version of the iterator class, IConstantIterator with the allElementsDo() function. The words print in the same order as they were originally added to Stack1.

Because of the nature of the stack class, the program must use the constant iterator class, IConstantIterator, when printing the stacks. It uses the push() and pop() functions for adding and removing elements, respectively. The allElementsDo() function is used when the collection is printed.

```
// pushpop.C  -  An example of using a Stack
#include <string.h>
#include <iostream.h>
                        // Let's use the default stack: IStack
#include <istack.h>

typedef IStack <char*> SimpleStack;
                        // The stack requires iteration to be const.
typedef IConstantIterator <char*> StackIterator;


// Test variables to put into our Stack:

char *String[9] = {    "The",  "quick",  "brown",  "fox",
                        "jumps",  "over",  "a",  "lazy",  "dog." };

// A class to display the contents of our Stack:

class PrintClass : public StackIterator
{
public:
   IBoolean applyTo(char* const& w)
      {
      cout << w << endl;
      return(True);
      }
};


// Main program
int main()
{
   SimpleStack Stack1, Stack2;
   char *S;
   PrintClass Print;

   // We specify two stacks.
   // First all the strings are pushed onto the first stack.
```

```
     // Next, they are popped from the first and pushed onto
     // the second.
     // Finally they are popped from the second and printed.
     // During this final print the strings must appear
     // in their original order.

     int i;

     for (i = 0; i < 9; i ++) {
        Stack1.push(String[i]);
        }

     cout << "Contents of Stack1:" << endl;
     Stack1.allElementsDo(Print);
     cout << "----------------------------" << endl;

     while (!Stack1.isEmpty()) {
        Stack1.pop(S);            // Pop from stack 1
        Stack2.push(S);           // Add it on top of stack 2
        }

     cout << "Contents of Stack2:" << endl;
     Stack2.allElementsDo(Print);
     cout << "----------------------------" << endl;

     while (!Stack2.isEmpty()) {
        Stack2.pop(S);
        cout << "Popped from Stack 2: " << S << endl;
        }

     return(0);
  }
```

This program produces the following output:

```
Contents of Stack1:
The
quick
brown
fox
jumps
over
a
lazy
dog.
----------------------------
Contents of Stack2:
dog.
lazy
a
over
jumps
fox
brown
quick
The
----------------------------
Popped from Stack 2: The
Popped from Stack 2: quick
Popped from Stack 2: brown
Popped from Stack 2: fox
Popped from Stack 2: jumps
Popped from Stack 2: over
Popped from Stack 2: a
Popped from Stack 2: lazy
Popped from Stack 2: dog.
```

Part 4. Tree Collection Classes

Tree Classes

Introduction to Trees

A tree is a collection of *nodes* that can have an arbitrary number of *references* to other nodes. There can be no cycles or short-circuit references. A unique path connects every two nodes. One node is designated as the *root* of the tree.

Formally, a tree can be defined recursively in the following manner:

1. A single *node* by itself is a tree. This node is also the *root* of the tree.
2. If N is a node and T-1, T-2, ..., T-k are trees with roots R-1, R-2, ..., R-k, respectively, then a new tree can be constructed by making N the *parent* of the nodes R-1, R-2, ..., R-k. In this new tree, N is the root and T-1, T-2, ..., T-k are the *subtrees* of the root N. Nodes R-1, R-2, ..., R-k are called *children* of node N.

Associated with each node is a data item called *element*.

Nodes without children are called *leaves* or *terminals*. The number of children in a node is called the *degree* of that node. The *level* of a given node is the number of steps in the path from the root to the given node. The root is at level 0 by definition. The *height* of a tree is the length of the longest path from the root to any node.

Defining the Traversal Order of Tree Elements

You can define the order in which nodes of a tree are traversed by specifying a parameter of type `ITreeIterationOrder` in calls to the following member functions:

- setToFirst
- setToLast
- setToNext
- setToPrevious
- allElementsDo, allSubtreeElementsDo

These functions are described in "N-ary Tree" on page 247.

The `ITreeIterationOrder` parameter can have one of two values: `IPreorder` or `IPostorder`. The effect of each of these values is explained below.

IPreorder The search begins at the root of the tree, and continues with the leftmost child of the root. If the child is the root of a subtree, the search continues with the leftmost child of the subtree, and so on, until a terminal node is detected. The search continues with all siblings of the terminal node, from left to right. If any of these siblings is the root of a subtree, the subtree is searched the same way as described above for the tree.

The preorder method can be summarized by the following recursive rules:

1. Visit the root.
2. Traverse the subtrees from left to right in preorder.

IPostorder The `IPostorder` method is the opposite of `IPreorder`. The search begins with the leftmost terminal node in the tree. Then that node's siblings are searched from left to right. If any of these siblings is the root of a subtree, the subtree is searched for its leftmost terminal node.

The postorder method can be sumarized by the following recursive rules:

1. Traverse the subtrees from left to right in postorder.
2. Visit the root.

The following figure shows a tree with 12 nodes, and the order of traversal for both preorder and postorder methods. Numbers indicate the preorder method (node 1 is searched before node 2) while letters indicate the postorder method (node A is searched before node B).

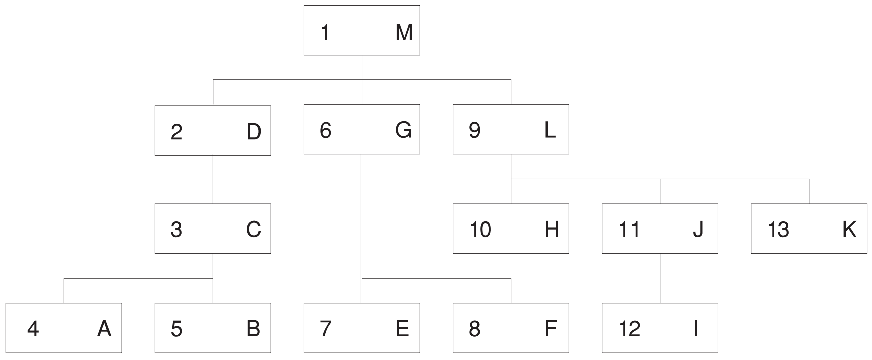


Figure 1. Preorder and Postorder Iteration Methods for Trees

N-ary Tree

An *n-ary tree* is a special tree where each node can have up to *n* children.

n must be greater than one. If *n* is one, the tree is a list. If *n* is zero, the structure loses its meaning.

An example of using an n-ary tree is a program used to build a family tree. (For simplicity, assume that the family tree is not concerned with information about spouses.) Whenever you discover a relative who is not already in your family tree, you enter the relative's name. If you know the parent's name, and the parent is already in the collection, the new relative is added as a child of the existing parent. If the parent is known but is not in the collection, a new collection is created, with the parent as the root element and the child as a child node of the parent. If you do not know the parent, the relative is entered as the root element of a new collection. You can also enter information about the children of a given relative; this information is used to attach a subtree, whose root node is the child, to the node of the parent of that child. Once you have established the collection, you can determine who is the parent or oldest known ancestor of a given relative, and you can display a list of all descendents of a given family member.

Derivation There are no bases or derived classes for N-ary Tree.

Variants and Header Files `ITree` is the default implementation variant based on tabular tree. `IGTree` is the default implementation variant with generic operations class. Both classes are declared in `itree.h`. No reference class exists for tree classes.

Members "Tree Functions" on page 251 lists the member functions for N-ary Tree.

Template Arguments and Required Functions

```
ITree  <Element, numberOfChildren>
IGTree <Element, StdOps, numberOfChildren>
```

The default implementation of `ITree` requires the following element functions:

Element Type

- Copy constructor
- Destructor
- Assignment

The argument value of `numberOfChildren()` specifies the maximum number of children for each node.

Terms Used

Some of the terms used in this chapter are defined below. You can also use the Glossary to look up terms you are unfamiliar with.

this tree	The tree to which a function is applied, in contrast to the *given tree*.
given ...	Referring to a tree, element, or function that is given as a function argument.
returned element	An element returned as a function return value.
iteration order	The order in which elements are visited in functions `allElementsDo()`, `allSubtreeElementsDo()`, `setToNext()`, and `setToPrevious()`.

Coding Example for N-ary Tree

The following sample constructs a binary tree for the following expression: *(8+2) * (2+4) / (7-5)*. The program prints this tree in preorder, using prefix notation. It then calculates the result of the expression. The program identifies subtrees consisting of an operand and two operators, calculates the result and replaces the subtree by its result. Finally, the tree consists of one node that is the result of the expression.

Note that the code does not respect precedence of "/" and "*" over "+" and "-".

```
// nary.C  -  An example of using an n-ary tree
#include <itree.h>
#include <istring.hpp>
#include <iostream.h>

/////////////////////////////////////////////////////////
// The tree for this expression is as follows:          //
//                                                      //
//                        /                             //
//                  *                -                  //
//              +        +      7         5             //
//             8   2     2   4                          //
/////////////////////////////////////////////////////////

typedef ITree <IString, 2> BinaryTree;

IBoolean printNode(IString const& node, void* dummy) {
// Prints one node of an n-ary tree
   cout << node << "|";
   return  True;
 }

void prefixedNotation(BinaryTree const& naryTree) {
// Prints an n-ary tree in prefixed notation
   naryTree.allElementsDo(printNode , IPreorder);
   cout << endl;
 }


void identifyChildren  (IString &child1,
                        IString &child2,
                        BinaryTree &binTree,
                        ITreeCursor &binTreeCursor) {
// Identifies the children of a node

  binTree.setToNext(binTreeCursor, IPreorder);
  child1 = binTree.elementAt(binTreeCursor);
  binTree.setToNextExistingChild(binTreeCursor);
  child2 = binTree.elementAt(binTreeCursor);
  binTree.setToParent(binTreeCursor);
 }


IBoolean isNumber(IString child) {
// Checks whether a node contains a number
  if ((child != '+') &&
      (child != '-') &&
      (child != '*') &&
      (child != '/'))
     { return True; }
  else { return False; }
}


void lookForNextOperator(BinaryTree &binTree,
                         ITreeCursor &binTreeCursor) {
// Looks for the next operator in the tree
   IBoolean operatorFound = False;

   do {
    if (!isNumber(binTree.elementAt(binTreeCursor))) {
         operatorFound = True;
       }
    else {
         binTree.setToNext(binTreeCursor, IPreorder);
       }
   }
   while (! operatorFound);
 }


void calculateSubtree(double &result, double &operand1,
                      double &operand2, IString &operatorSign) {
// Calculates the result from a subtree in the complete tree
   switch (*(char*)operatorSign) {
       case '+':
          result = operand1+operand2;
          break;
       case '-':
          result = operand1-operand2;
          break;
       case '/':
          result = operand1/operand2;
          break;
       case '*':
          result = operand1*operand2;
          break;
     } // end of switch
 }
```

```
/*********************** main ***************************/
int main () {
  // Construct the tree:

  BinaryTree  binTree;
  BinaryTree::Cursor binTreeCursor(binTree);
  BinaryTree::Cursor binTreeSaveCursor(binTree);


  binTree.addAsRoot("/");
  binTree.setToRoot(binTreeCursor);
  binTree.addAsChild(binTreeCursor, 1, "*");
  binTree.setToChild(1, binTreeCursor);
  binTree.addAsChild(binTreeCursor, 1, "+");
  binTree.setToChild(1, binTreeCursor);
  binTree.addAsChild(binTreeCursor, 1, "8");
  binTree.addAsChild(binTreeCursor, 2, "2");
  binTree.setToParent(binTreeCursor);
  binTree.addAsChild(binTreeCursor, 2, "+");
  binTree.setToChild(2, binTreeCursor);
  binTree.addAsChild(binTreeCursor, 1, "2");
  binTree.addAsChild(binTreeCursor, 2, "4");
  binTree.setToRoot(binTreeCursor);
  binTree.addAsChild(binTreeCursor, 2, "-");
  binTree.setToChild(2, binTreeCursor);
  binTree.addAsChild(binTreeCursor, 1, "7");
  binTree.addAsChild(binTreeCursor, 2, "5");

  // Print complete tree in prefix notation

  cout << "Printing the original tree in prefixed notation:"
      << endl;
  prefixedNotation(binTree);
  cout << " " << endl;

  // Calculate tree

  double      operand1 = 0;
  double      operand2 = 0;
  double      result = 0;
  INumber     replacePosition;
  IString     operatorSign, child1, child2;

  binTree.setToRoot(binTreeCursor);
  do
  {
   lookForNextOperator(binTree, binTreeCursor);
   operatorSign = binTree.elementAt(binTreeCursor);
   identifyChildren  (child1, child2, binTree, binTreeCursor);
   if ((isNumber(child1)) && (isNumber(child2)))
      {
        operand1 = child1.asDouble();
        operand2 = child2.asDouble();
        calculateSubtree(result, operand1, operand2,
                         operatorSign);
        if (binTree.numberOfElements() > 3)
        {
        // If tree contains more than three elements, replace
        // the calculated subtree by its result as follows.
        // (Save the cursor, because it will become invalidated after
        // removeSubtree)
         binTreeSaveCursor = binTreeCursor;
         binTree.setToParent(binTreeSaveCursor);
         replacePosition = binTree.position(binTreeCursor);
         binTree.removeSubtree(binTreeCursor);
```

```
           binTree.addAsChild(binTreeSaveCursor,
                              replacePosition,
                              (IString)result);
          cout << "Tree with calculated subtree replaced: "
               << endl;
          prefixedNotation(binTree);
          binTree.setToRoot(binTreeCursor);
          }
          else
          {
          // If tree contains root with two children only, replace
          // this calculated subtree by its result as follows:
           binTree.removeAll();
           binTree.addAsRoot(IString(result));
           cout << "Now the tree contains the result only:" << endl;
           prefixedNotation(binTree);
          }
        }
     else
        {
        binTree.setToNext(binTreeCursor, IPreorder);
        }
    }
    while (binTree.numberOfElements() > 1);

    return 0;
  }
```

The program produces the following output:

```
Printing the original tree in prefixed notation:
/|*|+|8|2|+|2|4|-|7|5|

Tree with calculated subtree replaced:
/|*|10|+|2|4|-|7|5|
Tree with calculated subtree replaced:
/|*|10|6|-|7|5|
Tree with calculated subtree replaced:
/|60|-|7|5|
Tree with calculated subtree replaced:
/|60|2|
Now the tree contains the result only:
30|
```

Tree Functions

This section lists the public member functions of n-ary trees.

Constructor `ITree ( ) ;`

Constructs a tree. The tree is initially empty; that is, it does not contain any nodes.

Copy Constructor **ITree** (ITree <*Element*, *numberOfChildren*> const& *tree*) ;

Constructs a tree by copying all elements from the given tree.

Exception: `IOutOfMemory`

Tree Collection Functions

Destructor

```
~ITree ( ) ;
```

Removes all elements from this tree.

Side Effects: All cursors of the tree become undefined.

operator=

```
ITree <Element, numberOfChildren>& operator= (
   ITree <Element, numberOfChildren> const& tree ) ;
```

Copies all elements of the given tree to this tree.

Return Value: A reference to this tree.

Side Effects: All cursors of this tree become undefined.

Exception: `IOutOfMemory`

addAsChild

```
void addAsChild ( ITreeCursor const& cursor,
   IPosition position, Element const& element ) ;
```

Adds the given element as a child with the given position to the node of this tree denoted by the given cursor.

Preconditions

- The cursor must point to an element of this tree.
- (1 ≤ *position* ≤ *numberOfChildren()*).
- The node denoted by the given cursor (of this tree) must not have a child at the given position.

Exceptions

- `IOutOfMemory`
- `ICursorInvalidException`
- `IPositionInvalidException`
- `IChildAlreadyExistsException`

Tree Collection Functions

addAsRoot

```
void addAsRoot ( Element const& element ) ;
```

Adds the given element as root of the tree.

Precondition: The tree must not have a root; that is, it must be empty.

Exceptions

- `IOutOfMemory`
- `IRootAlreadyExistsException`

allElementsDo, allSubtreeElementsDo

```
IBoolean allElementsDo (
   IBoolean (*function) (Element&, void*),
   ITreeIterationOrder iterationOrder,
   void* additionalArgument = 0 ) ;

IBoolean allElementsDo (
   IBoolean (*function) (Element const&, void*),
   ITreeIterationOrder iterationOrder,
   void* additionalArgument = 0 ) const;

IBoolean allSubtreeElementsDo ( ITreeCursor const& cursor,
   IBoolean (*function) (Element const&, void*),
   ITreeIterationOrder iterationOrder,
   void* additionalArgument = 0 ) const;

IBoolean allSubtreeElementsDo (
   ITreeCursor const& cursor,
   IBoolean (*function) (Element&, void*),
   ITreeIterationOrder iterationOrder,
   void* additionalArgument = 0 ) ;
```

Calls the given function for all elements of the subtree denoted by the given cursor (of this tree) until the given function returns `False`. The elements are visited in the given iteration order. Additional arguments can be passed to the given function using *`additionalArgument`*. The additional argument defaults to zero if no additional argument is given. The `allElementsDo()` function (without a subtree cursor argument) iterates over all elements of the tree.

Note: The given function must not remove elements from or add elements to the tree.

Return Value: Returns `True` if the given function returns `True` for every element it is applied to.

Preconditions

- The cursor must belong to this tree.
- The cursor must point to an element of this tree.

Exception: `ICursorInvalidException`

allElementsDo, allSubtreeElementsDo

```
IBoolean allElementsDo (
   IIterator <Element>& iterator,
   ITreeIterationOrder iterationOrder ) ;

IBoolean allElementsDo (
   IConstantIterator <Element>& iterator,
   ITreeIterationOrder iterationOrder ) const;


IBoolean allSubtreeElementsDo ( ITreeCursor const& cursor,
   IIterator <Element>& iterator,
   ITreeIterationOrder iterationOrder ) ;

IBoolean allSubtreeElementsDo ( ITreeCursor const& cursor,
   IConstantIterator <Element>& iterator,
   ITreeIterationOrder iterationOrder ) const;
```

Calls the `applyTo()` function of the given iterator for all elements of the subtree denoted by the given cursor (of this tree) until the `applyTo()` function returns `False`. The elements are visited in the given iteration order. The `allElementsDo()` function (without a subtree cursor argument) iterates over all elements of the tree.

Note: The `applyTo()` function must not remove elements from or add elements to the tree.

Preconditions

- The cursor must belong to this tree.
- The cursor must point to an element of this tree.

Return Value: Returns `True` if the `applyTo()` function returns `True` for every element it is applied to.

Exceptions: `ICursorInvalidException`

attachAsChild, attachSubtreeAsChild

```
void attachAsChild ( ITreeCursor const& cursor,
   IPosition position,
   ITree <Element, numberOfChildren>& tree ) ;

void attachSubtreeAsChild ( ITreeCursor const& cursor,
   IPosition position,
   ITree <Element, numberOfChildren>& tree,
   ITreeCursor const& subTreeCursor ) ;
```

Copies the subtree denoted by the given subtree cursor as a child with the given position of the node (of this tree) denoted by the given cursor. Removes this subtree from the given tree. The `attachAsChild()` function (without a subtree cursor argument) copies and removes the whole given tree.

Be careful when this tree and the given tree are the same. In such cases you must not attach a subtree to one of its own children, because a cyclic tree structure would result. Because `attachSubtreeAsChild()` removes this subtree from this tree, you will never be able to access either this subtree or the given subtree attached to it. This practice can also lead to memory not being properly freed.

This warning applies to both `attachAsChild()` and `attachSubtreeAsChild()`.

Note: These functions are implemented by copying a pointer to the subtree, rather than by copying all elements in the subtree.

Preconditions

- The cursor must point to an element of this tree.
- The subtree cursor must point to an element of the given tree.
- (1 ≤ *position* ≤ *numberOfChildren()*).
- The node denoted by the given cursor (of this tree) must not have a child at the given position.
- If this tree and the given tree are the same, a subtree must not be attached to one of its own children.

Exceptions

- `ICursorInvalidException`
- `IPositionInvalidException`
- `IChildAlreadyExistsException`
- `ICyclicAttachException`

Tree Collection Functions

attachAsRoot, attachSubtreeAsRoot

```
void attachAsRoot (
   ITree <Element, numberOfChildren>& tree ) ;

void attachSubtreeAsRoot (
   ITree <Element, numberOfChildren>& tree,
   ITreeCursor const& cursor ) ;
```

Copies the subtree denoted by the cursor of the given tree to (the root of) this tree, and removes this subtree from the given tree. The `attachAsRoot()` function (without a cursor argument) copies and removes the whole given tree.

Note: These functions are implemented by copying a pointer to the subtree, rather than by copying all elements in the subtree.

Preconditions

- The cursor must point to an element of this tree.
- The tree must not have a root; that is, it must be empty.

Exceptions

- `ICursorInvalidException`
- `IRootAlreadyExistsException`

copy, copySubtree

```
void copy (
   (ITree <Element, numberOfChildren> const& tree ) ;

void copySubtree (
   ITree <Element, numberOfChildren> const& tree,
   ITreeCursor const& cursor ) ;
```

Removes all elements from this tree, and copies the subtree denoted by the given cursor of the given tree to (the root of) this tree. The copy function (without a cursor argument) copies the whole given tree.

Preconditions: The cursor must point to an element of the given tree.

Exceptions

- `IOutOfMemory`
- `ICursorInvalidException`

elementAt

```
Element const& elementAt (
   ITreeCursor const& cursor ) const;
```

```
Element& elementAt ( ITreeCursor const& cursor ) ;
```

Returns a reference to the element pointed to by the given cursor.

Precondition: The cursor must point to an element of this tree.

Exception: `ICursorInvalidException`

hasChild

```
IBoolean hasChild ( IPosition position,
   ITreeCursor const& cursor ) const;
```

Returns `True` if the node pointed to by the given cursor has a child at the given position.

Preconditions

- The cursor must point to an element of this tree.
- (1 ≤ *position* ≤ *numberOfChildren()*)

Exceptions

- `ICursorInvalidException`
- `IPositionInvalidException`

isEmpty

```
IBoolean isEmpty ( ) const;
```

Returns `True` if the tree is empty.

isLeaf

```
IBoolean isLeaf ( ITreeCursor const& cursor ) const;
```

Returns `True` if the node pointed to by the given cursor is a leaf node of the tree. A leaf node is a node with no children.

Precondition: The cursor must point to an element of this tree.

Exception: `ICursorInvalidException`

Tree Collection Functions

isRoot

```
IBoolean isRoot ( ITreeCursor const& cursor ) const;
```

Returns True if the node pointed to by the given cursor is the root node of the tree.

Precondition: The cursor must point to an element of this tree.

Exception: ICursorInvalidException

newCursor

```
ITreeCursor* newCursor ( ) const;
```

Creates a cursor for the tree. The cursor is initially invalid.

Return Value: Pointer to the cursor.

Exception: IOutOfMemory

numberOfChildren

```
INumber numberOfChildren ( ) const;
```

Returns the number of children a node can possibly have. The actual number of children of any node will always be less than or equal to this number.

numberOfElements, numberOfSubtreeElements

```
INumber numberOfElements ( ) const;

INumber numberOfSubtreeElements (
   ITreeCursor const& cursor ) const;
```

Returns the number of elements that the subtree denoted by the given cursor contains. The subtree root, inner, and leaf nodes are counted. The numberOfElements() function (without a cursor argument) counts the number of elements in the whole tree.

Preconditions: The cursor must belong to the tree and must point to an element in the tree.

Exception: ICursorInvalidException

numberOfLeaves, numberOfSubtreeLeaves

```
INumber numberOfLeaves ( ) const;

INumber numberOfSubtreeLeaves (
   ITreeCursor const& cursor ) const;
```

Returns the number of leaf elements that the subtree denoted by the given cursor contains. Leaves are nodes that have no children. The `numberOfLeaves()` function (without a cursor argument) counts the number of leaves in the whole tree.

Preconditions: The cursor must belong to the tree and must point to an element in the tree.

Exception: `ICursorInvalidException`

position

```
INumber position (
   ITreeCursor const& cursor ) const;
```

Returns the position of the node pointed to by the given cursor as a child with respect to its parent node. The position of the root node is 1.

Precondition: The cursor must point to an element of this tree.

Exception: `ICursorInvalidException`

removeAll, removeSubtree

```
void removeAll ( ) ;

void removeSubtree ( ITreeCursor const& cursor ) ;
```

Removes the subtree denoted by the given cursor (of this tree). The `removeAll()` function (without a cursor argument) removes all elements from this tree.

Precondition: The cursor must point to an element of this tree.

Side Effects: For `removeSubtree()`, the given cursor is invalidated after removal.

Exception: `ICursorInvalidException`

Tree Collection Functions

replaceAt

```
void replaceAt ( ITreeCursor const& cursor,
   Element const& element ) ;
```

Replaces the element pointed to by the cursor with the given element.

Precondition: The cursor must point to an element of this tree.

Exception: `ICursorInvalidException`

setToChild

```
IBoolean setToChild ( IPosition position,
   ITreeCursor& cursor ) const;
```

Sets the cursor to the child with the given position of the node denoted by the given cursor (of this tree). Invalidates the cursor if this child does not exist.

Preconditions

- The cursor must point to an element of this tree.
- (1 ≤ *position* ≤ *numberOfChildren()*).

Return Value: Returns `True` if the child exists.

Exceptions

- `ICursorInvalidException`
- `IPositionInvalidException`

setToFirst

```
IBoolean setToFirst ( ITreeCursor& cursor,
   ITreeIterationOrder iterationOrder ) const;
```

Sets the cursor to the first node in the given iteration order. Invalidates the cursor if the tree is empty.

Precondition: The cursor must belong to this tree.

Return Value: Returns `True` if the tree is not empty.

Exception: `ICursorInvalidException`

setToFirstExistingChild

```
IBoolean setToFirstExistingChild (
   ITreeCursor& cursor ) const;
```

Sets the cursor to the first child of the node denoted by the given cursor (of this tree). Invalidates the cursor if the node has no child. A node with no child is a leaf node of the tree.

Preconditions: The cursor must point to an element of this tree.

Return Value: Returns `True` if the node has a child.

Exception: `ICursorInvalidException`

setToLast

```
IBoolean setToLast ( ITreeCursor& cursor,
   ITreeIterationOrder iterationOrder ) const;
```

Sets the cursor to the last node in the given iteration order. Invalidates the cursor if the tree is empty.

Precondition: The cursor must belong to this tree.

Return Value: Returns `True` if the tree is not empty.

Exception: `ICursorInvalidException`

setToLastExistingChild

```
IBoolean setToLastExistingChild (
   ITreeCursor& cursor ) const;
```

Sets the cursor to the last child of the node denoted by the given cursor (of this tree). Invalidates the cursor if the node has no child. A node with no child is a leaf node of the tree.

Precondition: The cursor must point to an element of this tree.

Return Value: Returns `True` if the node has a child.

Exception: `ICursorInvalidException`

Tree Collection Functions

setToNext

```
IBoolean setToNext ( ITreeCursor& cursor,
   ITreeIterationOrder iterationOrder ) const;
```

Sets the cursor to the next node in the given iteration order. Invalidates the cursor if there is no next node.

Precondition: The cursor must point to an element of this tree.

Return Value: Returns `True` if the given cursor does not point to the last node (in iteration order).

Exception: `ICursorInvalidException`

setToNextExistingChild

```
IBoolean setToNextExistingChild (
   ITreeCursor& cursor ) const;
```

Sets the cursor to the next existing sibling of the node denoted by the given cursor (of this tree). Invalidates the cursor if the node has no next sibling. A node with no next sibling is the last existing child of its parent.

Precondition: The cursor must point to an element of this tree.

Return Value: Returns `True` if the node has a next sibling.

Exception: `ICursorInvalidException`

setToParent

```
IBoolean setToParent ( ITreeCursor& cursor ) const;
```

Sets the cursor to the parent of the node denoted by the given cursor (of this tree). Invalidates the cursor if the node has no parent. A node with no parent is the root node of its tree.

Precondition: The cursor must point to an element of this tree.

Return Value: Returns `True` if the node has a parent.

Exception: `ICursorInvalidException`

setToPrevious

```
IBoolean setToPrevious ( ITreeCursor& cursor,
   ITreeIterationOrder iterationOrder ) const;
```

Sets the cursor to the previous node in the given iteration order. Invalidates the cursor if there is no previous node.

Precondition: The cursor must point to an element of this tree.

Return Value: Returns `True` if the given cursor does not point to the first node (in iteration order).

Exception: `ICursorInvalidException`

setToPreviousExistingChild

```
IBoolean setToPreviousExistingChild (
   ITreeCursor& cursor ) const;
```

Sets the cursor to the previous existing sibling of the node denoted by the given cursor (of this tree). Invalidates the cursor if the node has no previous sibling. A node with no previous sibling is the first existing child of its parent.

Precondition: The cursor must point to an element of this tree.

Return Value: Returns `True` if the node has a previous sibling.

Exception: `ICursorInvalidException`

setToRoot

```
IBoolean setToRoot ( ITreeCursor& cursor ) const;
```

Sets the cursor to the root node of the tree. Invalidates the cursor if the tree is empty (that is, if no root node exists).

Precondition: The cursor must belong to this tree.

Return Value: Returns `True` if the tree is not empty.

Exception: `ICursorInvalidException`

Tree Collection Functions

Part 5. Auxiliary Collection Classes

This part describes the abstract collection classes. The abstract classes are the base classes from which concrete collection classes and their implementation variants are derived.

Auxiliary Classes

Cursor

Each collection class defines its own nested cursor class. All of these cursor classes are derived from one of the following classes:

- `IElementCursor`
- `IOrderedCursor`

`IOrderedCursor` is derived from `IElementCursor`, and `IElementCursor` is in turn derived from `ICursor`. Only cursors of ordered collections are derived from `IOrderedCursor`. Cursors from unordered collections are derived from `IElementCursor`, and only know the member functions from `IElementCursor` and `ICursor`.

This chapter describes the general member functions of these three cursor classes as well as the specific member functions provided for specific collections. Because the cursor classes are all abstract classes, no objects of type `IOrderedCursor`, `IElementCursor`, or `ICursor` can be declared. You can obtain cursor objects by using the collection member `newCursor()`, or by defining a cursor of a specific collection cursor class. The `newCursor()` member creates a cursor of the collection to which it is applied.

The `newCursor()` member returns a pointer to the newly created cursor object.

Each cursor object is associated with a collection object. A cursor function merely calls the corresponding function for this collection. For example, `cursor.setToFirst()` is the same as `collection.setToFirst(cursor)`, where `collection` is the object associated with `cursor`.

Header File

The cursor classes are declared in `icursor.h`. Note that individual collection header files already include `icursor.h`; you do not need to include the file in your programs.

Members

The cursor classes define the following methods:

Method	Page	Method	Page
Constructor	268	operator==	269
copy	268	setToFirst	269
isValid	268	setToLast	269
invalidate	268	setToNext	269
element	268	setToPrevious	270
operator!=	268		

Public Member Functions

Constructor

```
Cursor ( Collection const& collection ) ;
```

Constructs the cursor and associates it with the given collection. The cursor is initially invalid. The name of the constructor is that of the nested cursor class.

copy

```
void copy (ICursor const& cursor) ;
```

Copies the given cursor to this cursor. This cursor now points to where the given cursor points.

Precondition: The given cursor and this cursor must refer to the same collection type.

Note: This precondition cannot be checked.

isValid

```
IBoolean isValid ( ) const;
```

Returns `True` if the cursor points to an element of the associated collection.

invalidate

```
void invalidate ( ) ;
```

Invalidates the cursor; that is, it no longer points to an element of the associated collection.

element

```
Element const& element ( ) const;
```

Returns a constant reference to the element of the associated collection to which the cursor points.

Precondition: The cursor must point to an element of the associated collection.

Exception: `ICursorInvalidException`

operator!=

```
IBoolean operator!= ( Cursor const& cursor ) const;
IBoolean operator!= ( ICursor const& cursor ) const;
```

Returns `True` if the cursor does not point to the same element (of the same collection) as the given cursor.

operator==

```
IBoolean operator== ( Cursor const& cursor ) const;
IBoolean operator== ( ICursor const& cursor ) const;
```

Returns `True` if the cursor points to the same element (of the same collection) as the given cursor.

setToFirst

```
IBoolean setToFirst ( ) ;
```

Sets the cursor to the first element of the associated collection in iteration order. Invalidates the cursor if the collection is empty (if no first element exists).

Return Value: Returns `True` if the associated collection is not empty.

setToLast

```
IBoolean setToLast ( ) ;
```

Sets the cursor to the last element of the associated collection in iteration order. Invalidates the cursor if the collection is empty (no last element exists). This function is only available for cursors of ordered collections. Returns `True` if the associated collection was not empty.

setToNext

```
IBoolean setToNext ( ) ;
```

Sets the cursor to the next element in the associated collection in iteration order. Invalidates the cursor if no more elements are left to be visited. Returns `True` if there was a next element.

Precondition: The cursor must point to an element of the associated collection.

Exception: `ICursorInvalidException`

setToPrevious

```
IBoolean setToPrevious ( ) ;
```

Sets the cursor to the previous element of the associated collection in iteration order. Invalidates the cursor if no such element exists. This function is only available for cursors of ordered collections.

Return Value: Returns `True` if a previous element exists.

Precondition: The cursor must point to an element of the associated collection.

Exception: `ICursorInvalidException`

Tree Cursor

For n-ary trees, cursors are used to point to nodes in the tree. Unlike cursors of flat collections, tree cursors stay defined when elements are added to the tree, or when elements other than the one pointed to are removed. Cursors are used in operations to access the element information stored in a node. They are also used to designate a subtree of the tree, namely the subtree whose root node the cursor points to.

As for flat collections, a distinction is made between the abstract base class `ITreeCursor`, and cursor classes local to the tree classes themselves. The local, or nested, cursor classes are derived from the abstract base class.

Header Files The declarations for `ITreeCursor` can be found in `itcursor.h`.

Members Tree Cursor defines the following member functions:

Method	Page	Method	Page
Constructor	271	setToFirstExistingChild	272
operator!=	271	setToLastExistingChild	273
operator==	272	setToNextExistingChild	273
element	272	setToParent	273
isValid	272	setToPreviousExistingChild	273
invalidate	272	setToRoot	274
setToChild	272		

Public Members of Tree Cursor

Constructor

```
Cursor ( Tree const& tree ) ;
```

Constructs the cursor and associates it with the given tree. The cursor is initially invalid.

operator!=

```
IBoolean operator!= ( Cursor const& cursor ) ;
```

Returns `True` if the cursor does not point to the same node of the same tree as the given cursor.

Tree Cursor

operator== `IBoolean operator== ( Cursor const& cursor ) ;`

Returns `True` if the cursor points to the same node of the same tree as the given cursor.

element `Element const& element ( ) ;`

Returns a reference to the element of the associated tree to which the cursor points.

Preconditions: The cursor must point to a node of the associated tree.

Exception: `ICursorInvalidException`

isValid `IBoolean isValid ( ) ;`

Returns `True` if the cursor points to a node of the associated tree.

invalidate `void invalidate ( ) ;`

Invalidates the cursor so that it no longer points to a node of the associated tree.

setToChild `IBoolean setToChild ( IPosition position ) ;`

Sets the cursor to the child node with the given position. If the child does not exist, the cursor is invalidated. If the child at the given position exists, `setToChild()` returns `True`.

Preconditions

- (1 ≤ *position* ≤ `numberOfChildren`).
- The cursor must point to a node of the associated tree.

Exceptions

- `IPositionInvalidException`
- `ICursorInvalidException`

setToFirstExistingChild
`IBoolean setToFirstExistingChild ( ) ;`

Sets the cursor to the first existing child of the associated tree. If the node pointed to by the cursor has no children (that is, if the node is a leaf) the cursor is invalidated. If the node pointed to by the cursor has a child, `setToFirstExistingChild()` returns `True`.

setToLastExistingChild

`IBoolean` **`setToLastExistingChild`** `( ) ;`

Sets the cursor to the last existing child of the associated tree. If the node pointed to by the cursor has no children (that is, if the node is a leaf) the cursor is invalidated. If the node pointed to by the cursor has a child, `setToLastExistingChild()` returns `True`.

setToNextExistingChild

`IBoolean` **`setToNextExistingChild`** `( ) ;`

Sets the cursor to the next existing *sibling* of the node to which the cursor points. If the node to which the cursor points is the last child of its parent, no next existing child exists and the cursor is invalidated.

Return Value: Returns `False` if a next existing child exists.

Preconditions: The cursor must point to a node of the associated tree.

Exception: `ICursorInvalidException`

setToParent

`IBoolean` **`setToParent`** `( ) ;`

Sets the cursor to the parent of the node pointed to by the cursor. If the cursor points to the root, the node has no parent, and the cursor is invalidated.

Return Value: Returns `True` if the node has a parent.

Preconditions: The cursor must point to a node of the associated tree.

Exception: `ICursorInvalidException`

setToPreviousExistingChild

`IBoolean` **`setToPreviousExistingChild`** `( ) ;`

Sets the cursor to the previous existing *sibling* of the node to which the cursor points. If the node to which the cursor points is the last child of its parent, no more children exist and the cursor is invalidated.

Return Value: Returns `True` if there was a previous child.

Precondition: The cursor must point to a node of the associated tree.

Exception: `ICursorInvalidException`

Tree Cursor

setToRoot

```
IBoolean setToRoot ( ) ;
```

Sets the cursor to the root of the associated tree. If the collection is empty (if no root element exists), the cursor is invalidated. Otherwise, `setToRoot()` returns `True`.

Iterator and Constant Iterator Classes

The classes `IIterator` and `IConstantIterator` define the interface for iterator objects. The redefinition of the function `applyTo()` defines the actions that are performed with the version of `allElementsDo()` that takes an iterator argument. (See "allElementsDo" on page 110 for more information on this function.) Iteration stops when `applyTo()` returns `False`.

The figure Iteration Using Iterators in the *Open Class Library User's Guide* explains the concepts and usage of iterations.

Derivation These classes do not derive from any other class.

Header File `iiter.h`

Members These classes define only one function, as a virtual function.

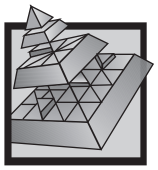
applyTo

```
virtual IBoolean applyTo (Element const& element) = 0;
```

This function applies a series of specified statements or a function to all elements of a collection for which you use the iterator. For example, `myCollection.allElementsDo(myIterator);` causes the code in the `applyTo()` function that you code for your iterator object `myIterator` to be applied to all elements of the collection `myCollection`.

For an example on how to use iterators, see "Iteration Using Iterators" on page 106 in the *Open Class Library User's Guide*.

Iterator and Constant Iterator Classes



Pointer Classes

The Collection Class Library defines five pointer classes:

- `IAutoPointer`
- `IAutoElemPointer`
- `IElemPointer`
- `IMngPointer`
- `IMngElemPointer`

These classes are declared in the header file `iptr.h`. You can select from these classes depending on your requirements:

- Pointers from classes named `I...ElemPointer` (also called **element pointers**) route the operations on the pointers to the referenced elements.
- Pointers from classes named `IAuto...Pointer` (also called **automatic pointers**) delete the elements they reference when the pointers are destructed. No reference count is kept.
- Pointers from classes named `IMng...Pointer` (also called **managed pointers**) keep a reference count for each referenced element. When the last managed pointer to the element is destructed, the element is automatically deleted.

For further information on the characteristics of these pointer types and how to use them, see "Using Pointer Classes" in the *Open Class Library User's Guide*.

Members

The pointer classes define constructors, a destructor, and four operators. An equality test operator, although not actually a member of the pointer classes, is also available.

Member	Page	Member	Page
Constructors	277	Conversion operator	278
Copy constructor	278	operator->	278
Destructors	278	operator=	278
operator*	278	operator==	279

Constructors

```
IAutoPointer ();
IElemPointer ();
IMngPointer ();
```

Constructs a pointer of the indicated type and initializes it with `NULL`.

Pointer Classes

Constructors from a Given C++ Pointer

```
IAutoPointer (Element *ptr, IExplicitInit)
IAutoElemPointer (Element *ptr, IExplicitInit)
IElemPointer (Element *ptr, IExplicitInit = IINIT)
IMngPointer (Element *ptr, IExplicitInit)
IMngElemPointer (Element *ptr, IExplicitInit)
```

Constructs a pointer object of the indicated type from a given C++ pointer. For managed pointers, the reference count of the referenced element is set to 1.

Copy Constructors from a Given Collection Class Pointer

```
IAutoPointer (IAutoPointer < Element > const& ptr)
IMngPointer (IMngPointer < Element > const& ptr)
```

Constructs a new pointer and initializes it with the given pointer. For automatic pointers, the given pointer is set to NULL. For managed pointers, the reference count of the referenced element is incremented by 1.

Destructors

```
~IAutoPointer ()
~IAutoElemPointer ()
```

Deletes the object referenced to by the automatic pointer.

```
~IMngPointer ()
~IMngElemPointer ()
```

Destructs the pointer and decrements the reference count of the referenced element. If the reference count is 0, the referenced element is deleted.

operator*

```
Element& operator * () const;
```

Returns a reference to the object to which the pointer refers.

Conversion operator

```
operator Element* () const
```

Implicitly convert this pointer to a C++ pointer.

operator->

```
Element* operator-> () const
```

Returns a C pointer to the object to which the pointer refers.

operator=

```
void                        operator = (IAutoPointer < Element > const& ptr)
IMngPointer < Element >&     operator = (IMngPointer < Element > const& ptr)
IMngElemPointer < Element >& operator = (IMngElemPointer < Element > const& ptr)
```

Assigns the given pointer to this pointer. For automatic pointers, the given pointer is set to `NULL` and the previously referenced element is deleted. For managed pointers, the reference count of the referenced element is incremented and the reference count of the previously referenced element is decremented.

operator== The pointer classes do not have an `operator==` explicitly defined for them. However, for equality test you can use the syntax:

```
pointerVariable1 == pointerVariable2;
```

The conversion operator (`operator Element*`) implicitly converts the objects to C pointers, and then the `operator==` for C pointers is invoked.

Because the `operator==` is not actually a member of the class, you cannot write an equality test like the following:

```
if (pointerVariable1.operator==(pointerVariable2)) {/* ... */}
```

Coding Example for Managed Element Pointer

The following sample allows you to store managed pointers for various graphical objects into a key sorted set. The graphical objects, namely lines, curves, and circles, inherit from a base class `Graphics`. Using these pointers, you can draw the various shapes from the collection.

```
// graph.C  - demonstrate how to use Collection Class pointers

#include <iostream.h>
#include "graph.h"
#include "line.h"
#include "circle.h"
#include "curve.h"
#include <iptr.h>
#include <iksset.h>

typedef IMngElemPointer <Graphics> MngGraphicsPointer;
typedef IKeySortedSet <MngGraphicsPointer, int> MngPointerKSet;

ostream & operator << (ostream & sout,
                       MngPointerKSet const& mgdPointerKSet) {
   MngGraphicsPointer drawObject;
   MngPointerKSet::Cursor
   gpsCursor(mgdPointerKSet);

   forCursor(gpsCursor) {
        drawObject = gpsCursor.element();

        sout << "\n Key is: " <<  drawObject->graphicsKey()
             << "\n ID is: " <<  drawObject->id() << endl;

        drawObject->draw();
     } /* endfor */

   return sout;
}
```

```
int main () {
    MngPointerKSet graphMngPointerKSet;
    //   Add curve pointers, circle pointers and line
    //   pointers to the graphMngPointerKSet.

     //Creating curve objects and adding pointers to the collections
     MngGraphicsPointer pcurve1 (new Curve
      (10, "Curve 1",
      1.1, 4.3,  2.1, 6.4,  3.1, 9.7,  4.1, 6.5,  5.1, 7.4),
      IINIT);
     MngGraphicsPointer pcurve2 (new Curve
      (20 ,"Curve 2",
      1.2, 3.9,  2.2, 5.9,  3.2, 8.8,  4.2, 7.5,  5.2, 9.4),
      IINIT);

     graphMngPointerKSet.add(pcurve1);
     graphMngPointerKSet.add(pcurve2);

     //Creating circle objects and adding pointers to the collections

     MngGraphicsPointer pcircle1 (new Circle
      (40 , "Circle 1" , 1.0, 1.0, 1.0), IINIT);
     MngGraphicsPointer pcircle2 (new Circle
      (50 , "Circle  2", 2.0, 2.0, 2.0), IINIT);

     graphMngPointerKSet.add(pcircle1);
     graphMngPointerKSet.add(pcircle2);

     //Creating line objects and adding pointers to the collections

     MngGraphicsPointer pline1 (new Line
      (70 , "Line 1" , 1.1 , 1.1 , 5.1 , 5.1), IINIT);
     MngGraphicsPointer pline2 (new Line
      (80 , "Line 2" , 2.2 , 2.2 , 5.2 , 5.2), IINIT);
     // if you want to have a normal C-pointer:
     Line* cPointerToLine = new Line
      (90 , "Line 3" , 3.3 , 3.3 , 5.3 , 5.3);
     MngGraphicsPointer pline3 (cPointerToLine, IINIT);

     graphMngPointerKSet.add(pline1);
     graphMngPointerKSet.add(pline2);
     graphMngPointerKSet.add(pline3);

     cout << "Drawing the shapes from the key set "
          << "of Managed Pointers: \n"
          << graphMngPointerKSet << "\n " << endl;

     graphMngPointerKSet.elementWithKey(70)->draw();
     cPointerToLine->draw();
     pline3->draw();

 // Now we are about to end the program.  The objects referenced
 // by managed pointers are automatically deleted.  See what
 // happens in the output of the program.
 return 0;
}
```

Part 6. Abstract Collection Classes

This part describes the abstract Collection Classes.

Abstract Classes

Collection

Derivation

Collection does not have any bases. Because collection is an abstract class, it cannot be used to create any objects. The following abstract classes are derived from collection:

- Key collection
- Equality collection
- Ordered collection

The concrete class heap is defined by collection.

The figure "The Abstract Class Hierarchy" in the *Open Class Library User's Guide* shows the relationship of collection to the class hierarchy.

Header File

Collection is declared in the header file `iacllct.h`.

Members

All the member functions of collection are defined as virtual functions and are described in "Introduction to Flat Collections" on page 97. The following member functions are provided for collection:

Method	Page	Method	Page
Destructor	101	isFull	118
add	103	maxNumberOfElements	123
addAllFrom	104	newCursor	123
anyElement	112	numberOfElements	123
copy	112	removeAll	125
elementAt	101	removeAt	126
elementAtPosition	115	replaceAt	128
isBounded	117	setToFirst	129
isEmpty	117	setToNext	130

Equality Collection

Because *equality collection* is an abstract class, it cannot be used to create any objects. The equality collection defines the interfaces for the property of element equality.

Derivation

Collection
  Equality Collection

The following abstract classes are derived from equality collection:

- Equality key collection
- Equality sorted collection

The following concrete classes are defined by equality collection:

- Set
- Bag
- Equality Sequence

The figure "The Abstract Class Hierarchy" in the *Open Class Library User's Guide* shows the relationship of equality collection to the class hierarchy.

Header File

The *equality collection* class is declared in the header file `iaequal.h`.

Members

The equality collection class defines the following member functions, described in "Introduction to Flat Collections" on page 97 , as virtual functions:

Method	Page	Method	Page
Destructor	101	locateOrAdd	121
contains	113	numberOfOccurrences	124
containsAllFrom	113	remove	125
locate	118	removeAllOccurrences	126
locateNext	120		

Equality Collection

Equality Key Collection

Because *equality key collection* is an abstract class, it cannot be used to create any objects. It defines the interfaces for the following properties:

- Element equality
- Key equality

Derivation

```
                    Collection
   Equality Collection          Key Collection
                Equality Key Collection
```

Equality key sorted collection is an abstract class that is derived from equality key collection. The following concrete classes are defined by equality key collection:

- Map
- Relation

The figure "The Abstract Class Hierarchy" in the *Open Class Library User's Guide* shows the relationship of equality key collection to the whole class hierarchy.

Header File The *equality key collection* class is declared in the header file `iaeqkey.h`.

Members All the members of equality key sorted collection are inherited from its base classes.

Equality Key Collection

Equality Key Sorted Collection

Equality key sorted collection is an abstract class that defines the interfaces for the following properties:

- Element equality
- Key equality
- Sorted elements

Because *equality key sorted collection* is an abstract class, it cannot be used to create any objects.

Derivation Equality key sorted collection is derived from the following three abstract classes:

- Key sorted collection
- Equality sorted collection
- Equality key sorted collection

For information on the bases of these classes, see the figure Figure 2 in the *Open Class Library User's Guide*.

The following concrete classes are defined by equality key sorted collection:

- Sorted map
- Sorted relation

The figure “The Abstract Class Hierarchy” in the *Open Class Library User's Guide* shows the relationship of equality key sorted collection to the class hierarchy.

Header File The *equality key sorted collection* class is declared in the header file `iaeqksrt.h`.

Members All the members of equality key sorted collection are inherited from its base classes.

Equality Key Sorted Collection

Equality Sorted Collection

Because *equality sorted collection* is an abstract class, it cannot be used to create any objects. It defines the interfaces for the following properties:

- Element equality
- Sorted elements

Derivation

```
                 Collection
                           Ordered Collection
  Equality Collection         Sorted Collection
          Equality Sorted Collection
```

Equality key sorted collection is an abstract class that is derived from equality sorted collection. The following concrete classes are defined by equality sorted collection:

- Sorted set
- Sorted bag

The figure "The Abstract Class Hierarchy" in the *Open Class Library User's Guide* shows the relationship of equality sorted collection to the class hierarchy.

Header File The *equality sorted collection* class is declared in the header file `iaeqsrt.h`.

Members All members of equality sorted collection are inherited from its base classes.

Key Collection

Because *key collection* is an abstract class, it cannot be used to create any objects. The key collection inherits from collection and defines the interfaces for the key property.

Derivation

Collection
  Key Collection

The following abstract classes are derived from key collection:

- Equality key collection
- Key sorted collection

The following concrete classes are defined by key collection:

- Key set
- Key bag

The figure “The Abstract Class Hierarchy” in the *Open Class Library User's Guide* shows the relationship of key collection to the class hierarchy.

Header File

The *key collection* class is declared in the header file `iakey.h`.

Members

The key collection class defines the following member functions, described in “Introduction to Flat Collections” on page 97 , as virtual functions:

Method	Page	Method	Page
Destructor	101	locateOrAddElementWithKey	122
addOrReplaceElementWithKey	109	numberOfDifferentKeys	123
containsAllKeysFrom	113	numberOfElementsWithKey	123
containsElementWithKey	113	removeAllElementsWithKey	126
elementWithKey	115	removeElementWithKey	127
key	118	replaceElementWithKey	129
locateElementWithKey	119	setToNextWithDifferentKey	130
locateNextElementWithKey	120		

Key Collection

Key Sorted Collection

Because *key sorted collection* is an abstract class, it cannot be used to create any objects. The key sorted collection inherits from sorted collection and key collection. It defines the interfaces for the following properties:

- Key equality
- Sorted elements

Derivation

```
                Collection
                        Ordered Collection
    Key Collection      Sorted Collection
               Key Sorted Collection
```

The equality key sorted collection is an abstract class that is derived from key sorted collection. The following concrete classes are defined by key sorted collection:

- Key sorted set
- Key sorted bag

The figure "The Abstract Class Hierarchy" in the *Open Class Library User's Guide* shows the relationship of key sorted collection to the class hierarchy.

Header File The *key sorted collection* class is declared in the header file `iaksrt.h`.

Members The key sorted collection class inherits all member functions from its base classes.

Key Sorted Collection

Ordered Collection

Because *ordered collection* is an abstract class, it cannot be used to create any objects. The ordered collection defines the interfaces for the property of ordered elements.

Derivation

Collection
  Ordered Collection

The following abstract classes are derived from ordered collection:

- Sorted collection
- Sequential collection

The figure "The Abstract Class Hierarchy" in the *Open Class Library User's Guide* shows the relationship of ordered collection to the class hierarchy.

Header File

The *ordered collection* class is declared in the header file `iaorder.h`.

Members

The ordered collection class defines the following member functions, described in "Introduction to Flat Collections" on page 97, as pure virtual functions:

Method	Page	Method	Page
Destructor	101	removeAtPosition	127
elementAtPosition	115	removeFirst	127
firstElement	117	removeLast	128
isFirst	117	setToLast	129
isLast	118	setToPosition	131
lastElement	118	setToPrevious	131

Ordered Collection

Sequential Collection

Because *sequential collection* is an abstract class, it cannot be used to create any objects. The sequential collection inherits from ordered collection and defines the interfaces for the properties of ordered elements.

Derivation

Collection
  Ordered Collection
    Sequential Collection

The following concrete classes are defined by sequential collection:

- Sequence
- Equality sequence

The figure "The Abstract Class Hierarchy" in the *Open Class Library User's Guide* shows the relationship of sequential collection to the class hierarchy.

Header File

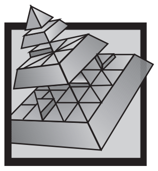
The *sequential collection* class is declared in the header file `iasqntl.h`.

Members

Sequential collection defines the following member functions as pure virtual functions:

Method	Page	Method	Page
Destructor	101	isFull	118
operator=	102	isLast	118
add	103	lastElement	118
addAllFrom	104	maxNumberOfElements	123
addAsFirst	105	newCursor	123
addAsLast	105	removeAll	125
addAsNext	106	removeAt	126
addAsPrevious	106	removeAtPosition	127
addAtPosition	107	removeFirst	127
allElementsDo	110	removeLast	128
anyElement	112	replaceAt	128
compare	112	setToFirst	129
elementAt	115	setToLast	129
elementAtPosition	115	setToNext	130
firstElement	117	setToPosition	131
isBounded	117	setToPrevious	131
isEmpty	117	sort	131
isFirst	117		

Sequential Collection



Sorted Collection

Because *sorted collection* is an abstract class, it cannot be used to create any objects. The sorted collection inherits from ordered collection and defines the interfaces for the properties of sorted elements.

Derivation

Collection
  Ordered Collection
    Sorted Collection

The following abstract classes are derived from sorted collection:

- Equality sorted collection
- Key sorted collection

The figure "The Abstract Class Hierarchy" in the *Open Class Library User's Guide* shows the relationship of sorted collection to the class hierarchy.

Header File

The *sorted collection* class is declared in the header file `iasrt.h`.

Members

The sorted collection class inherits all its members from its bases.

Sorted Collection

Part 7. Data Type and Exception Classes

The Data Type and Exception classes provide support for the exceptions, trace output, messages, strings, notifications, and window geometry used by the applications you develop.

Data Type and Exception Classes

Class Hierarchy

The data type and exception classes provide support for the exceptions, trace output, messages, strings, and notifications used by the applications you develop.

```
IBase
 ├─IBitFlag
 ├─IDate
 ├─INotificationEvent
 ├─IPair
 │  ├─IPoint
 │  ├─IRange
 │  └─ISize
 ├─IPointArray
 ├─IRectangle
 ├─IReference
 ├─IString
 │  └─I0String
 ├─IStringParser
 ├─ITime
 └─IVBase
    ├─IObserverList::Cursor
    ├─IBuffer
    │  └─IDBCSBuffer
    ├─IErrorInfo
    │  ├─ICLibErrorInfo
    │  ├─IGUIErrorInfo
    │  ├─IMMErrorInfo
    │  ├─ISystemErrorInfo
    │  └─IXLibErrorInfo
    ├─INotifier
    │  └─IStandardNotifier
    ├─IObserver
    ├─IObserverList
    ├─IRefCounted
    ├─IStringTest
    │  └─IStringTestMemberFn
    ├─ITrace
    └─IStringParser::SkipWords
IException
 ├─IAccessError
 ├─IAssertionFailure
 ├─IDeviceError
 ├─IInvalidParameter
 ├─IInvalidRequest
 └─IResourceExhausted
    ├─IOutOfMemory
    ├─IOutOfSystemResource
    └─IOutOfWindowResource
IExceptionLocation
IMessageText
IStringEnum
IException::TraceFn
IBase::Version
```

I0String

Derivation

IBase
  IString
    I0String

Inherited By None.

Header File i0string.hpp

Members

Member	Page	Member	Page
Constructor	308	lastIndexOf	314
adjustArg	317	lastIndexOfAnyBut	315
adjustResult	317	lastIndexOfAnyOf	315
change	310	notFound	318
charType	312	occurrencesOf	315
indexOf	313	operator []	312
indexOfAnyBut	313	overlayWith	312
indexOfAnyOf	314	remove	312
indexOfPhrase	316	subString	313
indexOfWord	316	~I0String	310
insert	311		

Objects of the I0String class are functionally equivalent to objects of the class IString (p. 469) with one major distinction: I0Strings are indexed starting at 0 instead of 1.

Note: A consequence of starting indexes at 0 is that you can no longer use the search functions as if they were Boolean. For example:

```
a0String.indexOf( anotherString ) != a0String.includes( anotherString ).
```

You can freely intermix IStrings and I0Strings in a program. You can assign objects of one class values of the other type. You can pass objects of either class as parameters to functions requiring the other type.

Warning: UINT_MAX is a reserved value for I0String. If you use UINT_MAX for the *startPos* parameter in I0String functions, unpredictable results can occur.

Public Functions

Constructors and Destructor

You can construct objects of this class in the following ways:

- Construct a NULL string.
- Construct a string with the ASCII representation of a given numeric value, supporting all variations of integer and double.
- Construct a string with a copy of the specified character data, supporting ASCIIZ strings, characters, and IStrings. The character data passed is converted to its ASCII representation.
- Construct a string with contents that consist of copies of up to three buffers of arbitrary data (void*). Optionally, you only need to provide the length, in which case the IString contents are initialized to a specified pad character. The default character is a blank.

These constructors can throw exceptions under the following conditions:

- Memory allocation errors

  Many factors dynamically allocate space and these allocation requests may fail. If so, the library translates memory allocation errors into exceptions. Generally, such errors do not occur until you allocate an astronomical amount of storage.

- Out-of-range errors

  These occur if you attempt to construct an IString with a length greater than UINT_MAX.

Constructors

1
```
I0String( const void* pBuffer1, unsigned lenBuffer1,
    char padCharacter = ' ');
```

Construct a string with contents from one buffer of arbitrary data (void*).

2
```
I0String();
```

Construct a NULL string.

3
```
I0String( const IString& aString);
```

Construct a string with a copy of the specified IString.

4
```
I0String( int);
```

Construct a string with the ASCII representation of an integer numeric value.

5
```
I0String( unsigned);
```

Construct a string with the ASCII representation of an unsigned numeric value.

6
```
I0String( long);
```

Construct a string with the ASCII representation of a long numeric value.

7
```
I0String( unsigned long);
```

Construct a string with the ASCII representation of an unsigned long numeric value.

8
```
I0String( short);
```

Construct a string with the ASCII representation of a short numeric value.

9
```
I0String( unsigned short);
```

Construct a string with the ASCII representation of an unsigned short numeric value.

10
```
I0String( double);
```

Construct a string with the ASCII representation of a double numeric value.

11
```
I0String( char);
```

Construct a string with a copy of the character. The string length is set to 1.

12
```
I0String( unsigned char);
```

Construct a string with a copy of the unsigned character. The string length is set to 1.

13
```
I0String( signed char);
```

Construct a string with a copy of the signed character. The string length is set to 1.

14
```
I0String( const char*);
```

Construct a string with a copy of the specified ASCIIZ string.

15
```
I0String( const unsigned char*);
```

Construct a string with a copy of the specified unsigned ASCIIZ string.

16
```
I0String( const signed char*);
```

Construct a string with a copy of the specified signed ASCIIZ string.

17
```
I0String( const void* pBuffer1, unsigned lenBuffer1,
    const void* pBuffer2, unsigned lenBuffer2,
    char padCharacter = ' ');
```

Construct a string with contents from two buffers of arbitrary data (void*).

18
```
I0String( const void* pBuffer1, unsigned lenBuffer1,
    const void* pBuffer2, unsigned lenBuffer2,
    const void* pBuffer3, unsigned lenBuffer3,
    char padCharacter = ' ');
```

Construct a string with contents from three buffers of arbitrary data (void*).

Destructor `virtual ~I0String();`

Editing

These members are reimplemented to treat the position arguments as 0-based.

change Changes occurrences of a specified pattern to a specified replacement string. You can specify the number of changes to perform. The default is to change all occurrences of the pattern. You can also specify the position in the receiver at which to begin.

The parameters are the following:

inputString
: The pattern string as a reference to an object of type IString. The library searches for the pattern string within the receiver's data.

pInputString
: The pattern string as NULL-terminated string. The library searches for the pattern string within the receiver's data.

outputString
: The replacement string as a reference to an object of type IString. It replaces the occurrences of the pattern string in the receiver's data.

pOutputString
: The replacement string as a NULL-terminated string. It replaces the occurrences of the pattern string in the receiver's data.

startPos The position to start the search at within the receiver's data. The default is 0.

numChanges
: The number of patterns to search for and change. The default is UINT_MAX, which causes changes to all occurrences of the pattern.

```
I0String& change( const IString& aPattern,
    const IString& aReplacement,
    unsigned startPos = 0,
    unsigned numChanges = ( unsigned ) UINT_MAX);

I0String& change( const IString& aPattern,
    const char* pReplacement, unsigned startPos = 0,
    unsigned numChanges = ( unsigned ) UINT_MAX);

I0String& change( const char* pPattern,
    const IString& aReplacement,
    unsigned startPos = 0,
    unsigned numChanges = ( unsigned ) UINT_MAX);
```

```
I0String& change( const char* pPattern,
    const char* pReplacement, unsigned startPos = 0,
    unsigned numChanges = ( unsigned ) UINT_MAX);

static I0String change( const IString& aString,
    const IString& inputString,
    const IString& outputString,
    unsigned startPos = 0,
    unsigned numChanges = ( unsigned ) UINT_MAX);

static I0String change( const IString& aString,
    const IString& inputString,
    const char* pOutputString,
    unsigned startPos = 0,
    unsigned numChanges = ( unsigned ) UINT_MAX);

static I0String change( const IString& aString,
    const char* pInputString,
    const IString& outputString,
    unsigned startPos = 0,
    unsigned numChanges = ( unsigned ) UINT_MAX);

static I0String change( const IString& aString,
    const char* pInputString,
    const char* pOutputString,
    unsigned startPos = 0,
    unsigned numChanges = ( unsigned ) UINT_MAX);
```

insert

Inserts the specified string at the specified location.

```
static I0String insert( const IString& aString,
    const IString& anInsert,
    unsigned index = ( unsigned ) UINT_MAX,
    char padCharacter = ' ');

I0String& insert( const IString& aString,
    unsigned index = ( unsigned ) UINT_MAX,
    char padCharacter = ' ');

I0String& insert( const char* pString,
    unsigned index = ( unsigned ) UINT_MAX,
    char padCharacter = ' ');

static I0String insert( const IString& aString,
    const char* pInsert,
    unsigned index = ( unsigned ) UINT_MAX,
    char padCharacter = ' ');
```

overlayWith Replaces a specified portion of the receiver's contents with the specified string. The overlay starts in the receiver's data at the *index*, which defaults to 0. If *index* is beyond the end of the receiver's data, it is padded with the pad character (*padCharacter*).

```
static I0String overlayWith( const IString& aString,
    const IString& anOverlay, unsigned index = 0,
    char padCharacter = ' ');

I0String& overlayWith( const IString& aString,
    unsigned index = 0, char padCharacter = ' ');

I0String& overlayWith( const char* pString,
    unsigned index = 0, char padCharacter = ' ');

static I0String overlayWith( const IString& aString,
    const char* pOverlay, unsigned index = 0,
    char padCharacter = ' ');
```

remove Deletes the specified portion of the string (that is, the substring) from the receiver. You can use this function to truncate an IString object at a specific position. For example:

```
aString.remove(8);
```

removes the substring beginning at index 8 and takes the rest of the string as a default.

```
I0String& remove( unsigned startPos);
I0String& remove( unsigned startPos, unsigned numChars);
static I0String remove( const IString& aString, unsigned startPos);
static I0String remove( const IString& aString,
    unsigned startPos, unsigned numChars);
```

Queries

These members are overridden to permit specification of the index as a 0-based value.

charType Returns the type of the character at the specified index.

```
IStringEnum::CharType charType( unsigned index) const;
```

operator [] Returns a reference to the specified character of the string.

Note: If you call the non-const version of this function with an index beyond the end, the function extends the string.

```
const char& operator []( unsigned index) const;
char& operator []( unsigned index);
```

subString Returns the specified portion of the string (that is, the substring) of the receiver.

The parameters are the following:

startPos The starting position of the substring being extracted. If this position is beyond the end of the data in the receiver, this function returns a NULL IString.

length The length of the substring to be extracted. If the length extends beyond the end of the receiver's data, the returned IString is padded to the specified length with *padCharacter*. If you do not specify *length* and it defaults, this function uses the rest of the receiver's data starting from *startPos* for padding.

padCharacter
The character the function uses as padding if the requested length extends beyond the end of the receiver's data. The default *padCharacter* is a blank.

You can use this function to truncate an IString object at a specific position. For example:

```
aString = aString.subString(0, 7);
```

returns the substring concluding with index 7 and discards the rest of the string.

```
I0String subString( unsigned startPos) const;
I0String subString( unsigned startPos, unsigned len,
    char padCharacter = ' ') const;
```

Searches

These members are reimplemented to treat the starting position of the search as a 0-based index.

indexOf Returns the byte index of the first occurrence of the specified string within the receiver. If there are no occurrences, 0 is returned. In addition to IStrings, you can also specify a single character or an IStringTest (p. 515).

```
unsigned indexOf( const IString& aString, unsigned startPos = 0) const;
unsigned indexOf( const char* pString, unsigned startPos = 0) const;
unsigned indexOf( char aCharacter, unsigned startPos = 0) const;
unsigned indexOf( const IStringTest& aTest,
   unsigned startPos = 0) const;
```

indexOfAnyBut
Returns the index of the first character of the receiver that is not in the specified set of characters. If there are no characters, 0 is returned. Alternatively, this function returns the index of the first character that fails the test prescribed by a specified IStringTest (p. 515) object.

```
unsigned indexOfAnyBut( const IStringTest& aTest,
    unsigned startPos = 0) const;

unsigned indexOfAnyBut( const IString& aString,
    unsigned startPos = 0) const;

unsigned indexOfAnyBut( const char* pValidChars,
    unsigned startPos = 0) const;

unsigned indexOfAnyBut( char validChar,
    unsigned startPos = 0) const;
```

indexOfAnyOf

Returns the index of the first character of the receiver that is a character in the specified set of characters. If there are no characters, 0 is returned. Alternatively, this function returns the index of the first character that passes the test prescribed by a specified IStringTest (p. 515) object.

```
unsigned indexOfAnyOf( char searchChar, unsigned startPos = 0) const;

unsigned indexOfAnyOf( const IString& searchChars,
    unsigned startPos = 0) const;

unsigned indexOfAnyOf( const char* pSearchChars,
    unsigned startPos = 0) const;

unsigned indexOfAnyOf( const IStringTest& aTest,
    unsigned startPos = 0) const;
```

lastIndexOf

Returns the index of the last occurrence of the specified string or character. The search starts at the position specified by *startPos* (inclusive) and proceeds backward. The returned value is in the range *0 <= x <= startPos* or I0String::notFound. The default of UINT_MAX-1 starts the search at the end of the receiver's string. If the search target is not found, 0 is returned.

If you specify 0 for *startPos*, the search starts at the beginning of the string. Therefore, because the search proceeds backward from its starting position, in this case the search target must occur at the beginning of the string for it to be found.

```
unsigned lastIndexOf( char aCharacter,
    unsigned endPos = ( unsigned ) ( UINT_MAX - 1 )) const;

unsigned lastIndexOf( const IString& aString,
    unsigned endPos = ( unsigned ) ( UINT_MAX - 1 )) const;

unsigned lastIndexOf( const char* pString,
    unsigned endPos = ( unsigned ) ( UINT_MAX - 1 )) const;

unsigned lastIndexOf( const IStringTest& aTest,
    unsigned startPos = ( unsigned ) ( UINT_MAX - 1 )) const;
```

lastIndexOfAnyBut

Returns the index of the last character not in the specified string or character. The search starts at the position specified by *startPos* (inclusive) and proceeds backward. The default of UINT_MAX-1 starts the search at the end of the receiver's string. If the search target is not found, 0 is returned.

If you specify 0 for *startPos*, the search starts at the beginning of the string. Therefore, because the search proceeds backward from its starting position, in this case the search target must occur at the beginning of the string for it to be found.

```
unsigned lastIndexOfAnyBut( const IString& validChars,
    unsigned endPos = ( unsigned ) ( UINT_MAX - 1 )) const;

unsigned lastIndexOfAnyBut( const char* pValidChars,
    unsigned endPos = ( unsigned ) ( UINT_MAX - 1 )) const;

unsigned lastIndexOfAnyBut( char validChar,
    unsigned startPos = ( unsigned ) ( UINT_MAX - 1 )) const;

unsigned lastIndexOfAnyBut( const IStringTest& aTest,
    unsigned endPos = ( unsigned ) ( UINT_MAX - 1 )) const;
```

lastIndexOfAnyOf

Returns the index of the last character in the specified string or character. The search starts at the position specified by *startPos* (inclusive) and proceeds backward. The default of UINT_MAX-1 starts the search at the end of the receiver's string. If the search target is not found, 0 is returned.

If you specify 0 for *startPos*, the search starts at the beginning of the string. Therefore, because the search proceeds backward from its starting position, in this case the search target must occur at the beginning of the string for it to be found.

```
unsigned lastIndexOfAnyOf( const IString& searchChars,
    unsigned endPos = ( unsigned ) ( UINT_MAX - 1 )) const;

unsigned  lastIndexOfAnyOf( const char* pSearchChars,
    unsigned endPos = ( unsigned ) ( UINT_MAX - 1 )) const;

unsigned lastIndexOfAnyOf( char searchChar,
    unsigned startPos = ( unsigned ) ( UINT_MAX - 1 )) const;

unsigned lastIndexOfAnyOf( const IStringTest& aTest,
    unsigned endPos = ( unsigned ) ( UINT_MAX - 1 )) const;
```

occurrencesOf

Returns the number of occurrences of the specified IString, char*, char, or IStringTest. If you just want a Boolean test, this is slower than IString::indexOf (p. 481).

```
unsigned occurrencesOf( const IStringTest& aTest,
    unsigned startPos = 0) const;

unsigned occurrencesOf( const IString& aString,
    unsigned startPos = 0) const;

unsigned occurrencesOf( const char* pString,
    unsigned startPos = 0) const;

unsigned occurrencesOf( char aCharacter, unsigned startPos = 0) const;
```

Word Operations

These members are reimplemented to treat the result index as 0-based.

indexOfPhrase

Returns the position of the first occurrence of the specified phrase in the receiver. If the phrase is not found, 0 is returned.

```
unsigned indexOfPhrase( const IString& wordString,
    unsigned startWord = 1) const;
```

indexOfWord

Returns the index of the specified white-space-delimited word in the receiver. If the word is not found, 0 is returned.

```
unsigned indexOfWord( unsigned wordNumber) const;
```

Inherited Public Functions

IString		
asDebugInfo	isASCII	operator =
asDouble	isBinaryDigits	operator char *
asInt	isControl	operator signed char *
asString	isDBCS	operator unsigned char *
asUnsigned	isDigits	operator []
b2c	isGraphics	operator ^
b2d	isHexDigits	operator ^=
b2x	isLike	operator |
c2b	isLowerCase	operator |=
c2d	isMBCS	operator ˜
c2x	isPrintable	overlayWith
center	isPunctuation	**remove**
change	isSBCS	**removeWords**

IString		
charType	isUpperCase	reverse
copy	isValidDBCS	**rightJustify**
d2b	isValidMBCS	size
d2c	isWhiteSpace	space
d2x	lastIndexOf	**strip**
includes	lastIndexOfAnyBut	**stripBlanks**
includesDBCS	lastIndexOfAnyOf	stripLeading
includesMBCS	**leftJustify**	**stripLeadingBlanks**
includesSBCS	length	stripTrailing
indexOf	lengthOfWord	**stripTrailingBlanks**
indexOfAnyBut	**lineFrom**	subString
indexOfAnyOf	lowerCase	translate
indexOfPhrase	numWords	**upperCase**
indexOfWord	occurrencesOf	word
insert	operator &	wordIndexOfPhrase
isAbbreviationFor	operator &=	words
isAlphabetic	operator +	**x2b**
isAlphanumeric	operator +=	**x2c**

IBase		
asDebugInfo	**messageFile**	**setMessageFile**
asString	**messageText**	**version**

Protected Functions

Index Mapping

Use these members to convert arguments and results to the proper index base: 1 for arguments (because it relies on IString) and 0 for results (because it is 0 based itself).

adjustArg Adjusts the specified index from 0- to 1-based.

```
static unsigned adjustArg( unsigned index);
```

adjustResult Adjusts a function result from 1- to 0-based.

```
static unsigned adjustResult( unsigned index);
```

Inherited Protected Functions

IString		
applyBitOp	findPhrase	isLike
buffer	indexOfWord	**lengthOf**
change	initBuffer	occurrencesOf
data	insert	overlayWith
defaultBuffer	isAbbrevFor	setBuffer

Public Data

Searches

These members are reimplemented to treat the starting position of the search as a 0-based index.

notFound You use this static constant in conjunction with the searching functions. It specifies the value searching functions return indicating the search failed.

```
static const unsigned notFound;
```

Inherited Protected Data

IString		
maxLong	**null**	**nullBuffer**

IBase		
recoverable	**unrecoverable**	

IAccessError

Derivation

IException
  IAccessError

Inherited By None.

Header File iexcbase.hpp

Members

Member	Page
Constructor	319
name	320

Objects of the IAccessError class represent an exception. When a member function makes a request of the operating system or the presentation system that the system cannot satisfy, the member function creates and throws an object of the IAccessError class. If the operating system or the presentation system cannot satisfy the request due to resource exhaustion, member functions create and throw objects of the class IResourceExhausted (p. 459).

Note: Typically, if no other exception fits an error condition, the User Interface Class Library creates and throws an object of the IAccessError class.

Public Functions

Constructor

You can construct objects of this class.

Constructor You can create objects of this class by doing the following:

- Using the constructor.

  errorText The text describing this particular error.

  errorId The identifier you want to associate with this particular error.

  severity Use the enumeration IException::Severity (p. 386) to specify the severity of the error. The default is unrecoverable.

- Using the macros discussed in IException (p. 379). The User Interface Class Library provides these macros to make creating exceptions easier for you.

```
IAccessError( const char* errorText, unsigned long errorId,
   Severity severity = IException::unrecoverable);
```

Exception Type

Exception Type members provide support for determining the name (type) of the exception. This is used for logging out an exception object's error information.

name Returns the name of the object's class.

```
virtual const char* name() const;
```

Inherited Public Functions

IException		
addLocation	locationAtIndex	setSeverity
appendText	locationCount	setText
assertParameter	logExceptionData	**setTraceFunction**
errorCodeGroup	name	terminate
errorId	setErrorCodeGroup	text
isRecoverable	setErrorId	textCount

Inherited Public Data

IException		
baseLibrary	**CLibrary**	**operatingSystem**

IAssertionFailure

Derivation IException
  IAssertionFailure

Inherited By None.

Header File iexcbase.hpp

Members

Member	Page
Constructor	321
name	322

Objects of the IAssertionFailure class represent an exception. The IASSERT macro expands to create and throw an object of the IAssertionFailure class if the specified condition is not met. An assertion is a debugging tool you can use to assure a condition is true. The class IException (p. 379) describes IASSERT and the other exception-handling macros.

Public Functions

Constructor

You can construct objects of this class.

Constructor You can create objects of this class by doing the following:

- Using the constructor.

  errorText The text describing this particular error.

  errorId The identifier you want to associate with this particular error.

  severity Use the enumeration IException::Severity (p. 386) to specify the severity of the error. The default is unrecoverable.

- Using the macro IASSERT (p. 379). The User Interface Class Library provides this macro to make creating exceptions easier for you.

```
IAssertionFailure( const char* errorText, unsigned long errorId,
   Severity severity = IException::unrecoverable);
```

Exception Type

Exception Type members provide support for determining the name (type) of the exception. This is used for logging out an exception object's error information.

name Returns the name of the object's class.

```
virtual const char* name() const;
```

Inherited Public Functions

IException		
addLocation	locationAtIndex	setSeverity
appendText	locationCount	setText
assertParameter	logExceptionData	**setTraceFunction**
errorCodeGroup	name	terminate
errorId	setErrorCodeGroup	text
isRecoverable	setErrorId	textCount

Inherited Public Data

IException		
baseLibrary	**CLibrary**	**operatingSystem**

IBase

Derivation Inherits from none.

Inherited By

IAccelerator
IBitFlag
ICnrAllocator
ICoordinateSystem
ICritSec
IDate
IDDEActiveServer
IEventData
IEventParameter1
IEventParameter2
IEventResult
IFileDialog::Settings
IFontDialog::Settings
IFrameExtension
IGraphicBundle
IHandle
IHelpWindow::Settings
IHighEventParameter
ILowEventParameter
IMenuItem
IMMAudioBuffer
INotificationEvent
IPair
IPointArray
IProcedureAddress
IRectangle
IReference
IResourceId
IString
IStringGenerator
IStringParser
IStringParser::SkipWords
ISWP
ISWPArray
ITime
ITransformMatrix
IVBase

Header File ibase.hpp

Members

Member	Page	Member	Page
asDebugInfo	324	recoverable	325
asString	324	setMessageFile	325
messageFile	324	unrecoverable	325
messageText	324	version	324

The IBase class encapsulates the set of names that otherwise would be in global scope. All the classes in the library inherit from this class. Thus, you can use the types and enumeration values defined here in those classes without the qualifying IBase:: prefix.

Other code, not within the scope of IBase, must use either the qualified names or the simplified synonyms which the User Interface Class Library declares in ISYNONYM.HPP.

Public Functions

Diagnostics

Use these members to provide diagnostic information.

asDebugInfo This function obtains the diagnostic version of an object's contents. You use this to retrieve an IString representing a hex pointer to the object.

```
IString asDebugInfo() const;
```

asString This function obtains the standard version of an object's contents.

```
IString asString() const;
```

version Returns the User Interface Class Library version using the major and minor data members of the IBase::Version data structure. The minor number is incremented to indicate the service level. This is a static member function.

```
static Version version();
```

Messages

Use these members to query and change the message file that contains text used by the class library when throwing exceptions.

messageFile This function returns the name of the message file used to load library exception text.

```
static char* messageFile();
```

PM If you previously called setMessageFile (p. 325) with the name of a message file, the file's name is returned. Otherwise, the library checks the environment variable ICLUI MSGFILE for the message file name. You can set the environment variable using:

```
SET ICLUI MSGFILE=mymsgfile.msg
```

You must specify the file extension, typically .MSG. If you have not set the environment variable, the library uses the default message file (CPPOOC3U.MSG).

Motif If you previously called setMessageFile (p. 325) with the name of a message file, the file's name is returned. The default file name is ibmcl.cat.

messageText Returns the message text associated with the specified message ID. You can specify up to nine optional text strings to insert into the message.

```
static IMessageText messageText( unsigned long messageId,
    const char* textInsert1 = 0, const char* textInsert2 = 0,
    const char* textInsert3 = 0, const char* textInsert4 = 0,
    const char* textInsert5 = 0, const char* textInsert6 = 0,
    const char* textInsert7 = 0, const char* textInsert8 = 0,
    const char* textInsert9 = 0);
```

PM If the message is found in a message segment that has been bound to the .EXE, the message is loaded from the application. Otherwise, the message is searched for in the message file described before. The search order for this file is as follows:

- The system root directory
- The current working directory
- Using the DPATH environment setting
- Using the APPEND environment setting

Motif The AIX release uses the message catalog file ibmcl.cat. You must add the /nls subdirectory to your NLSPATH environment variable so the User Interface Class Library can access the message catalog. In the Korn Shell, put the following statement in your .profile file:

```
export NLSPATH=<targetdir>/nls/%N:$NLSPATH
```

where <targetdir> is the directory in which you installed the User Interface Class Library.

setMessageFile

Sets the message file from which the class library loads its exception text. The name must include the file extension.

```
static void setMessageFile( const char* msgFileName);
```

Protected Data

Exception Severity

These data members are provided as synonyms for the IException::Severity enumeration, which is used when constructing an IException object or an object of one of its derived classes.

recoverable Synonym for IException::recoverable. Use this when constructing an IException object or one of its subclasses:

```
static IException::Severity recoverable;
```

unrecoverable

Synonym for IException::unrecoverable. Use this when constructing an IException object or one of its subclasses:

```
static IException::Severity unrecoverable;
```

Nested Type Definitions

BooleanConstants

```
BooleanConstants { false = 0, true = 1 };
```

The User Interface Class Library provides this enumeration to define constant values for false and true. Never use true for an equality test because you should consider any nonzero value to be true. This constant provides a useful mnemonic for setting a Boolean.

Boolean

```
typedef int Boolean;
```

General true or false type used as an argument or return value for many member functions.

IBase::Version

Derivation Inherits from none.

Inherited By None.

Header File ibase.hpp

Members

Member	Page
major	327
minor	327

This structure (data type) defines the version specifier, comprised of major and minor version numbers.

Public Data

major The major version level of the library. It is incrememnted by 1 for each new release within a version.

```
unsigned short major;
```

minor The minor version level of the library. It starts at 0 for each version level and is incrememnted by 1 for each CSD.

```
unsigned short minor;
```

IBitFlag

Derivation

IBase
  IBitFlag

Inherited By

I3StateCheckBox::Style
IAnimatedButton::Style
IBaseComboBox::Style
IBaseListBox::Style
IBaseSpinButton::Style
IBitmapControl::Style
IButton::Style
ICanvas::Style
ICheckBox::Style
ICircularSlider::Style
IComboBox::Style
IContainerControl::Attribute
IContainerControl::Style
IControl::Style
ICustomButton::Style
IDMImage::Style
IDrawingCanvas::Style
IEntryField::Style
IFileDialog::Style
IFontDialog::Style
IFrameWindow::Style
IGraphicPushButton::Style
IGroupBox::Style
IIconControl::Style
IListBox::Style
IListBoxDrawItemHandler::DrawFlag
IMenu::Style
IMenuBar::Style
IMenuDrawItemHandler::DrawFlag
IMenuItem::Attribute
IMenuItem::Style
IMessageBox::Style
IMultiCellCanvas::Style
IMultiLineEdit::Style
INotebook::clrFlags
INotebook::PageSettings::Attribute
INotebook::Style
INumericSpinButton::Style
IOutlineBox::Style
IProgressIndicator::Style
IPushButton::Style
IRadioButton::Style
IScrollBar::Style
ISetCanvas::Style
ISlider::Style
ISpinButton::Style
ISplitCanvas::Style
IStaticText::Style
ITextSpinButton::Style
IToolBar::Style
IToolBarButton::Style
IToolBarContainer::Style
IViewPort::Style
IWindow::Style

Header File

ibitflag.hpp

Members

Member	Page	Member	Page
Constructor	332	operator !=	331
asExtendedUnsignedLong	331	operator ==	331
asUnsignedLong	331	setValue	331

The IBitFlag class is the abstract base class for the bitwise styles and attributes used by window and control classes in the User Interface Class Library. Because this class in an abstract base class, you cannot create objects of this class.

Deriving Classes from IBitFlag

Typically, you can declare classes derived from IBitFlag by using the macros that accompany this class. Optionally, these macros lets you:

- Construct objects of one derived class from objects of another class derived from IBitFlag.
- Combine objects of one derived class with objects of another class derived from IBitFlag. For example, using the bitwise OR operator.

Macro Descriptions

You can use the following macros to declare classes derived from IBitFlag:

INESTEDBITFLAGCLASSDEF0 macro
Declares a logical base bitwise flag class scoped to another class. A logical base bitwise class is a class of bitwise flag objects that cannot be constructed from a bitwise flag object of another class.

INESTEDBITFLAGCLASSDEF1 macro
Declares a bitwise flag class whose objects can be constructed from an object of another bitwise flag class. The library assumes both bitwise flag classes have the same name and are scoped to another class.

INESTEDBITFLAGCLASSDEF2 macro
Declares a bitwise flag class whose objects can be constructed from an object of two other bitwise flag classes. The library assumes all the bitwise flag classes have the same name and are scoped to another class.

INESTEDBITFLAGCLASSDEF3 macro
Declares a bitwise flag class whose objects can be constructed from an object of three other bitwise flag classes. The library assumes all the bitwise flag classes have the same name and are scoped to another class.

INESTEDBITFLAGCLASSDEF4 macro
Declares a bitwise flag class whose objects can be constructed from an object of four other bitwise flag classes. The library assumes all the bitwise flag classes have the same name and are scoped to another class.

INESTEDBITFLAGCLASSFUNCS macro
Declares global functions that operate on a class of bitwise flag objects scoped to another class. The functions are global, rather than member functions, to allow for commutative operations between objects of different bitwise flag classes.

Note: Do not use this macro to define global functions for a logical base style class (one declared with the INESTEDBITFLAGCLASSDEF0 macro).

Public Functions

Comparisons

Use these members to compare bitflag values.

operator != Used to compare two bitflag values for inequality.

```
Boolean operator !=( const IBitFlag& rhs) const;
```

operator == Used to compare two bitflag values for equality.

```
Boolean operator ==( const IBitFlag& rhs) const;
```

Queries

Use these members to return the value of the object.

asExtendedUnsignedLong

Converts the upper 32-Bits of the object to an unsigned long value.

```
unsigned long asExtendedUnsignedLong() const;
```

asUnsignedLong

Converts the object to an unsigned long value.

```
unsigned long asUnsignedLong() const;
```

Inherited Public Functions

IBase		
asDebugInfo	**messageFile**	**setMessageFile**
asString	**messageText**	**version**

Protected Functions

Assignment

Use this member to set the value of the object.

setValue You can use this function to assign an unsigned long value for the system styles to the object, and optionally an unsigned long value that represents extended styles.

```
IBitFlag& setValue( unsigned long value,
    unsigned long extendedValue = 0);
```

Constructor

You can only construct objects of this class from an unsigned long value, which represents the styles accepted by the system, and, optionally, an unsigned long value that represents extended styles.

Note: This constructor is protected because objects derived from this class should not be arbitrarily constructed. To provide type safety for window and control constructors, you can only specify the following:

- Existing style objects
- Existing attribute objects
- Combinations of these objects

Constructor `IBitFlag( unsigned long value, unsigned long extendedValue = 0);`

Inherited Protected Data

IBase		
recoverable	**unrecoverable**	

IBuffer

Derivation

IBase
  IVBase
    IBuffer

Inherited By

IDBCSBuffer

Header File

ibuffer.hpp

Members

Member	Page
Constructor	344
addRef	339
allocate	344
asDebugInfo	334
center	334
change	334
charType	337
checkAddition	338
checkMultiplication	338
className	345
compare	334
contents	337
copy	335
dbcsTable	345
defaultBuffer	337
fromContents	337
includesDBCS	336
includesMBCS	336
includesSBCS	336
indexOf	340
indexOfAnyBut	340
indexOfAnyOf	340
initialize	344
insert	335
isAlphabetic	342
isAlphanumeric	342
isASCII	342
isControl	342
isDBCS	337
isDigits	342
isGraphics	342
isHexDigits	343
isLowerCase	343
isMBCS	337
isPrintable	343
isPunctuation	343
isSBCS	337
isUpperCase	343
isValidDBCS	337
isValidMBCS	337
isWhiteSpace	343
lastIndexOf	340
lastIndexOfAnyBut	341
lastIndexOfAnyOf	341
leftJustify	335
length	337
lowerCase	335
newBuffer	338
next	338
null	338
operator delete	344
operator new	344
overflow	339
overlayWith	335
remove	335
removeRef	339
reverse	335
rightJustify	336
setDefaultBuffer	339
startBackwardsSearch	345
startSearch	345
strip	336

Member	Page	Member	Page
subString	341	useCount	338
translate	336	˜IBuffer	344
upperCase	336		

Objects of the IBuffer class define the contents of an IString (p. 469).

Public Functions

Comparisons

Use these members to compare the IBuffer's contents to some other character array.

compare Compares the buffer’s contents to the contents of the specified character array.

```
virtual Comparison compare( const void* p, unsigned len) const;
```

Diagnostics

Use these members to provide diagnostic information about the buffer.

asDebugInfo Returns information about the buffer’s internal representation that you can use for debugging.

```
virtual IString asDebugInfo() const;
```

Editing

These members are called by the corresponding IString members to edit the buffer's contents.

center Centers the receiver within a string of the specified length.

```
virtual IBuffer* center( unsigned newLen, char padCharacter);
```

change Changes occurrences of a specified pattern to a specified replacement string. You can specify the number of changes to perform. The default is to change all occurrences of the pattern. You can also specify the position in the receiver at which to begin.

The parameters are the following:

pSource The pattern string as NULL-terminated string. The library searches for the pattern string within the receiver’s data.

sourceLen The length of the source string.

pTarget The target string as a NULL-terminated string. It replaces the occurrences of the pattern string in the receiver’s data.

targetLen The length of the target string.

startPos The position to start the search at within the target's data.

numChanges
The number of patterns to search for and change.

```
virtual IBuffer* change( const char* pSource,
    unsigned sourceLen, const char* pTarget,
    unsigned targetLen, unsigned startPos,
    unsigned numChanges);
```

copy Replaces the receiver's contents with a specified number of replications of itself.

```
virtual IBuffer* copy( unsigned numCopies);
```

insert Inserts the specified string after the specified location.

```
virtual IBuffer* insert( const char* pInsert,
    unsigned insertLen, unsigned pos, char padCharacter);
```

leftJustify Left-justifies the receiver in a string of the specified length. If the new length (*newLen*) is larger than the current length, the string is extended by the pad character (*padCharacter*). The default pad character is a blank.

```
virtual IBuffer* leftJustify( unsigned newLen, char padCharacter);
```

lowerCase Translates all upper-case letters in the receiver to lower-case.

```
virtual IBuffer* lowerCase();
```

overlayWith Replaces a specified portion of the receiver's contents with the specified string. If *pos* is beyond the end of the receiver's data, it is padded with the pad character (*padCharacter*).

```
virtual IBuffer* overlayWith( const char* overlay,
    unsigned len, unsigned pos, char padCharacter);
```

remove Deletes the specified portion of the string (that is, the substring) from the receiver. You can use this function to truncate an IString object at a specific position. For example:

```
aString.remove(8);
```

removes the substring beginning at index 8 and takes the rest of the string as a default.

```
virtual IBuffer* remove( unsigned startPos, unsigned numChars);
```

reverse Reverses the receiver's contents.

```
virtual IBuffer* reverse();
```

rightJustify Right-justifies the receiver in a string of the specified length. If the receiver's data is shorter than the requested length (*newLen*), it is padded on the left with the pad character (*padCharacter*). The default pad character is a blank.

```
virtual IBuffer* rightJustify( unsigned newLen, char padCharacter);
```

strip Strips both leading and trailing character or characters. You can specify the character or characters as the following:

- A char* array
- An IStringTest (p. 515) object

The default is white space.

```
virtual IBuffer* strip( const IStringTest& aTest,
    IStringEnum::StripMode mode);

virtual IBuffer* strip( const char* pChars, unsigned len,
    IStringEnum::StripMode mode);
```

translate Converts all of the receiver's characters that are in the first specified string to the corresponding character in the second specified string.

```
virtual IBuffer* translate( const char* pInputChars,
    unsigned inputLen, const char* pOutputChars,
    unsigned outputLen, char padCharacter);
```

upperCase Translates all lower-case letters in the receiver to upper-case.

```
virtual IBuffer* upperCase();
```

NLS Testing

Corresponding IString members use these members to test the buffer's contents. These tests are character-set-specific.

includesDBCS

If any characters are DBCS (double-byte character set), true is returned.

```
virtual Boolean includesDBCS() const;
```

includesMBCS

If any characters are MBCS (multiple-byte character set), true is returned.

```
virtual Boolean includesMBCS() const;
```

includesSBCS

If any characters are SBCS (single-byte character set), true is returned.

```
virtual Boolean includesSBCS() const;
```

isDBCS If all the characters are DBCS, true is returned.

```
virtual Boolean isDBCS() const;
```

isMBCS If all the characters are MBCS, true is returned.

```
virtual Boolean isMBCS() const;
```

isSBCS If all the characters are SBCS, true is returned.

```
virtual Boolean isSBCS() const;
```

isValidDBCS If no DBCS characters have a 0 second byte, true is returned.

```
virtual Boolean isValidDBCS() const;
```

isValidMBCS If no MBCS characters have a 0 second byte, true is returned.

```
virtual Boolean isValidMBCS() const;
```

Queries

Use these members to access various attributes of a buffer.

charType Returns the type of a character at the specified index.

```
virtual IStringEnum::CharType charType( unsigned index) const;
```

contents Returns the address of the buffer's contents.

```
const char* contents() const;
char* contents();
```

defaultBuffer Returns the address of the NULL buffer for the class. This is a static function.

```
static IBuffer* defaultBuffer();
```

fromContents

Returns the address of IBuffer using the specified pointer to its contents. This is a static function.

Note: It is important that *pBuffer* point to the actual beginning of data from an IBuffer object. The library can return only values from the contents function of this class. Otherwise, if the returned IBuffer pointer is used, errors could occur.

```
static IBuffer* fromContents( const char* pBuffer);
```

length Returns the length of the buffer's contents.

```
unsigned length() const;
```

next Returns a pointer to the next character, not the next byte, in the buffer.

```
virtual char* next( const char* prev);
virtual const char* next( const char* prev) const;
```

null Returns the address of the NULL buffer.

```
IBuffer* null() const;
```

useCount Returns the number of IStrings referring to the buffer.

```
unsigned useCount() const;
```

Reallocation

Use these members to manage reallocation of IBuffers when the contents of strings are modified.

checkAddition

Verifies that the two parameters, when added, do not overflow an unsigned integer.

```
static unsigned checkAddition( unsigned addend1, unsigned addend2);
```

checkMultiplication

Verifies that the two parameters, when multiplied, do not overflow an unsigned integer.

```
static unsigned checkMultiplication( unsigned factor1, unsigned factor2);
```

newBuffer Allocates a new buffer and initializes it with the contents of up to three specified buffers.

The parameters are the following:

p1	The pointer to the first part to be copied into the data area of the new buffer. The first part is *len1* bytes long. If the pointer is NULL, the *padChar* is copied for *len1* bytes.
len1	The length, in bytes, of the first part to be copied into the new buffer.
p2	A pointer to the second part, immediately following the first part, to be copied into the data area of the new buffer. The second part is *len2* bytes long. If the pointer is NULL, the *padChar* is copied for *len2* bytes. If nothing is specified for *p2*, it is NULL.
len2	The length, in bytes, of the second part to be copied into the new buffer. If nothing is specified for *len2*, it defaults to 0 bytes.

p3	The pointer to the third part, immediately following the second part, to be copied into the data area of the new buffer. The third part is *len3* bytes long. If the pointer is NULL, the *padChar* is copied for *len3* bytes. If nothing is specified for *p3*, it is NULL.
len3	The length, in bytes, of the third part to be copied into the new buffer. If nothing is specified for *len3*, it defaults to 0 bytes.
padChar	The character to use as the pad in the cases of *p1*, *p2*, or *p3* being NULL. If you do not specify a *padChar*, it defaults to the character 0.

Note: If the sum of *len1*, *len2*, and *len3* is 0, a reference to the NULL buffer for this class is added and the address is returned.

```
IBuffer* newBuffer( const void* p1, unsigned len1,
    const void* p2 = 0, unsigned len2 = 0,
    const void* p3 = 0, unsigned len3 = 0,
    char padChar = 0) const;
```

overflow Throws an exception when IBuffer::checkAddition (p. 338) or IBuffer::checkMultiplication (p. 338) detect an overflow.

```
static unsigned overflow();
```

Exception

IInvalidRequest. You made an IBuffer request causing an overflow. Typically, this occurs during object construction or during an operation which grows an underlying IBuffer object. Likely culprits might be an IBuffer::newBuffer or IString::change call.

setDefaultBuffer

Sets the default (NULL) buffer. The specified buffer must be comprised of a single NULL byte.

```
static void setDefaultBuffer( IBuffer* newDefaultBuffer);
```

Reference Counting

Use these members to manage the buffer reference count.

addRef Increments the usage count.

```
void addRef();
```

removeRef Decrements the usage count and deletes the buffer when the usage count goes to 0.

```
void removeRef();
```

Searches

These members are called by the corresponding IString members to search the buffer's contents.

indexOf Returns the byte index of the first occurrence of the specified string within the receiver. If there are no occurrences, 0 is returned.

```
virtual unsigned indexOf( const char* pString,
    unsigned len, unsigned startPos = 1) const;

virtual unsigned indexOf( const IStringTest& aTest,
    unsigned startPos = 1) const;
```

indexOfAnyBut

Returns the index of the first character of the receiver that is not in the specified set of characters. If there are no characters, 0 is returned. Alternatively, this function returns the index of the first character that fails the test prescribed by a specified IStringTest (p. 515) object.

```
virtual unsigned indexOfAnyBut( const IStringTest& aTest,
    unsigned startPos = 1) const;

virtual unsigned indexOfAnyBut( const char* pString, unsigned len,
    unsigned startPos = 1) const;
```

indexOfAnyOf

Returns the index of the first character of the receiver that is a character in the specified set of characters. If there are no characters, 0 is returned. Alternatively, this function returns the index of the first character that passes the test prescribed by a specified IStringTest (p. 515) object.

```
virtual unsigned indexOfAnyOf( const char* pString,
    unsigned len, unsigned startPos = 1) const;

virtual unsigned indexOfAnyOf( const IStringTest& aTest,
    unsigned startPos = 1) const;
```

lastIndexOf Returns the index of the last occurrence of the specified string or character. The search starts at the position specified by *startPos* (inclusive) and proceeds backward. The returned value is in the range *0 <= x <= startPos*. The default of 0 starts the search at the end of the receiver's string. If the search target is not found, 0 is returned.

If you specify 0 for *startPos*, this function returns 0 indicating the search target was not found.

```
virtual unsigned lastIndexOf( const char* pString,
    unsigned len, unsigned startPos = 0) const;

virtual unsigned lastIndexOf( const IStringTest& aTest,
    unsigned startPos = 0) const;
```

lastIndexOfAnyBut

Returns the index of the last character not in the specified string or character. The search starts at the position specified by *startPos* (inclusive) and proceeds backward. The default of 0 starts the search at the end of the receiver's string. If the search target is not found, 0 is returned.

If you specify 0 for *startPos*, this function returns 0 indicating the search target was not found.

```
virtual unsigned lastIndexOfAnyBut( const IStringTest& aTest,
    unsigned startPos = 0) const;

virtual unsigned lastIndexOfAnyBut( const char* pString,
    unsigned len, unsigned startPos = 0) const;
```

lastIndexOfAnyOf

Returns the index of the last character in the specified string or character. The search starts at the position specified by *startPos* (inclusive) and proceeds backward. The default of 0 starts the search at the end of the receiver's string. If the search target is not found, 0 is returned.

If you specify 0 for *startPos*, this function returns 0 indicating the search target was not found.

```
virtual unsigned lastIndexOfAnyOf( const IStringTest& aTest,
    unsigned startPos = 0) const;

virtual unsigned lastIndexOfAnyOf( const char* pString,
    unsigned len, unsigned startPos = 0) const;
```

Subset

Use this member when a subset of characters is required.

subString Returns a new IBuffer, of the same type as the previous one, containing the specified subset of characters.

The parameters are the following:

startPos The index at which to start the substring. If *startPos* is 0, the function uses position 1. If *startPos* is beyond the end of the buffer, nothing is copied. The buffer is filled out by the specified padding character.

len The length to copy from the buffer. If the length extends beyond the end of the buffer, only the portion up to the end is copied. The buffer is then padded. If *len* is 0, a reference to the NULL buffer is returned.

padCharacter
Specifies the character the function uses to pad the copied string if less than *len* bytes have been copied from the source buffer.

```
virtual IBuffer* subString( unsigned startPos,
    unsigned len, char padCharacter) const;
```

Testing

Corresponding IString members use these members to test the buffer's contents.

isAlphabetic If all the characters are in {'A'-'Z','a'-'z'}, true is returned.

```
virtual Boolean isAlphabetic() const;
```

isAlphanumeric
If all the characters are in {'A'-'Z','a'-'z','0'-'9'}, true is returned.

```
virtual Boolean isAlphanumeric() const;
```

isASCII If all the characters are in {0x00-0x7F}, true is returned.

```
virtual Boolean isASCII() const;
```

isControl Returns true if all the characters are control characters.

Control characters are defined by the iscntrl() C Library function as defined in the cntrl locale source file and in the cntrl class of the LC_CTYPE category of the current locale. For example, on ASCII operating systems, control characters are those in the range {0x00-0x1F,0x7F}.

```
virtual Boolean isControl() const;
```

isDigits If all the characters are in {'0'-'9'}, true is returned.

```
virtual Boolean isDigits() const;
```

isGraphics Returns true if all the characters are graphics characters.

Graphics characters are printable characters excluding the space character, as defined by the isgraph() C Library function in the graph locale source file and in the graph class of the LC_CTYPE category of the current locale. For example, on ASCII operating systems, graphics characters are those in the range {0x21-0x7E}.

```
virtual Boolean isGraphics() const;
```

isHexDigits

If all the characters are in {'0'-'9','A'-'F','a'-'f'}, true is returned.

```
virtual Boolean isHexDigits() const;
```

isLowerCase

If all the characters are in {'a'-'z'}, true is returned.

```
virtual Boolean isLowerCase() const;
```

isPrintable

Returns true if all the characters are printable characters.

Printable characters are defined by the isprint() C Library function as defined in the print locale source file and in the print class of the LC_CTYPE category of the current locale. For example, on ASCII systems, printable characters are those in the range {0x20-0x7E}.

```
virtual Boolean isPrintable() const;
```

isPunctuation

If none of the characters is white space, a control character, or an alphanumeric character, true is returned.

```
virtual Boolean isPunctuation() const;
```

isUpperCase

If all the characters are in {'A'-'Z'}, true is returned.

```
virtual Boolean isUpperCase() const;
```

isWhiteSpace

Returns true if all the characters are whitespace characters.

Whitespace characters are defined by the isspace() C Library function as defined in the space locale source file and in the space class of the LC_CTYPE category of the current locale. For example, on ASCII systems, printable characters are those in the range {0x09-0x0D,0x20}.

```
virtual Boolean isWhiteSpace() const;
```

Inherited Public Functions

IVBase		
asDebugInfo	asString	

IBase		
asDebugInfo	**messageFile**	**setMessageFile**
asString	**messageText**	**version**

Protected Functions

Allocation

Use these protected members to allocate and deallocate IBuffer objects.

allocate

Returns a new buffer of the specified length.

```
virtual IBuffer* allocate( unsigned bufLength) const;
```

operator delete

Deallocates a buffer.

```
void operator delete( void* p, const char* filename, size_t linenum);
void operator delete( void* p);
```

operator new

Allocates space for a buffer of the specified length. The returned pointer is an area the size of an IBuffer large enough to hold data of size *bufLen*.

```
void* operator new( size_t t, const char* filename,
    size_t linenum, unsigned bufLen);

void* operator new( size_t t, unsigned bufLen);
```

Constructor and Destructor

Constructors for this class require the length of the buffer, which is the value to be stored in the len data member.

Constructor

```
IBuffer( unsigned newLen);
```

Initializes the reference count and null terminates the buffer.

Destructor

```
~IBuffer();
```

Destructor, does nothing.

Implementation

This member helps implement this class.

initialize

Initializes (sets up a NULL buffer, a DBCS table, and so forth). This is a static function.

```
static IBuffer* initialize();
```

Protected Queries

This member helps implement this class.

className Returns the name of the class (IBuffer).

```
virtual const char* className() const;
```

Search Initialization

These members help implement this class.

startBackwardsSearch

Initializes a search of type IString::lastIndexOf (p. 489).

- If *searchLen* is greater than the length of the buffer, 0 is returned indicating an invalid search request.
- If the starting position is 0 or beyond the last *searchLen* bytes of the buffer, the position where the last *searchLen* bytes start in the buffer is returned.
- If the starting position is 1 through the last *searchLen* bytes, the value of *startingPos* is returned.

```
virtual unsigned startBackwardsSearch( unsigned startPos,
    unsigned searchLen) const;
```

startSearch Initializes a search of type IString::indexOf (p. 481).

- If *startPos* is 0, the search uses a starting position of 1.
- If the specified *startPos* and *searchLen* result in an invalid search, 0 is returned. This usually occurs when the sum of *startPos* and *searchLen* is greater than the size of the buffer.

```
virtual unsigned startSearch( unsigned startPos,
    unsigned searchLen) const;
```

Public Data

DBCS Table

Use this character array member to test characters for DBCS validity.

dbcsTable Table of DBCS first-byte flags ('dbcsTable[n] == 1' if and only if n is a valid DBCS first byte).

```
static char dbcsTable [ 256 ];
```

Supported On:
PM

Inherited Protected Data

IBase		
recoverable	**unrecoverable**	

Nested Type Definitions

Comparison

```
typedef enum { equal , greaterThan , lessThan } Comparison;
```

These enumerators specify the possible valid return codes from IBuffer::compare (p. 334).

equal
The buffer's contents are equal to the contents of the specified character array.

greaterThan
The buffer's contents are greater than the contents of the specified character array.

lessThan
The buffer's contents are less than the contents of the specified character array.

ICLibErrorInfo

Derivation

IBase
  IVBase
    IErrorInfo
      ICLibErrorInfo

Inherited By None.

Header File iexcept.hpp

Members

Member	Page	Member	Page
Constructor	348	text	348
errorId	348	throwCLibError	349
isAvailable	348	˜ICLibErrorInfo	348
operator const char *	348		

Objects of the ICLibErrorInfo class represent error information. When a C library call results in an error condition, objects of the ICLibErrorInfo class are created. The per thread global variable errno is used to obtain the error text.

The library provides the ITHROWCLIBERROR macro for throwing exceptions constructed with ICLibErrorInfo information. This macro has the following parameters:

location	The name of the C function returning the error code, the name of the file the function is in, and the function's line number.
name	Use the enumeration IErrorInfo::ExceptionType (p. 377) to specify the type of the exception. The default is accessError.
severity	Use the enumeration IException::Severity (p. 386) to specify the severity of the error. The default is recoverable.

This macro generates code that calls throwCLibError (p. 349), which does the following:

1. Creates an ICLibErrorInfo object
2. Uses the object to create an IException object
3. Adds the CLibrary error group to the object

4. Adds location information
5. Logs the exception data
6. Throws the exception

Public Functions

Constructor and Destructor

You can construct and destruct objects of this class. You cannot copy or assign objects of this class.

Constructor

```
ICLibErrorInfo( const char* CLibFunctionName = 0);
```

You can only construct objects of this class using the default constructor.

Note: If the constructor cannot load the error text, the library provides the following default text: "No error text is available."

CLibFunctionName
(Optional) The name of the failing C library function. If you specify *CLibFunctionName*, the constructor prefixes it to the error text.

Destructor

```
virtual ~ICLibErrorInfo();
```

Error Information

Use these members to return the error information provided by objects of this class.

errorId
Returns the value of errno, which you can use to obtain the errno information.

```
virtual unsigned long errorId() const;
```

isAvailable
If the error text is available, true is returned.

```
virtual Boolean isAvailable() const;
```

operator const char *
Returns the error text.

```
virtual operator const char *() const;
```

text
Returns the error text.

```
virtual const char* text() const;
```

Throw Support

Use these members to support the throwing of exceptions.

throwCLibError

Creates an ICLibErrorInfo object and uses the text from it to the following:

1. Create an exception object
2. Add the location information to it
3. Log the exception data
4. Throw the exception

functionName
The name of the function where the exception occurred.

location An IExceptionLocation (p. 389) object containing the following:

- Function name
- File name
- Line number where the function is called

name Use the enumeration IErrorInfo::ExceptionType (p. 377) to specify the type of the exception. The default is accessError.

severity Use the enumeration IException::Severity (p. 386) to specify the severity of the error. The default is recoverable.

```
static void throwCLibError( const char* functionName,
    const IExceptionLocation& location,
    IErrorInfo::ExceptionType name = accessError,
    IException::Severity severity = recoverable);
```

Inherited Public Functions

IErrorInfo		
errorId	isAvailable	operator const char *

IVBase		
asDebugInfo	asString	

IBase		
asDebugInfo	**messageFile**	**setMessageFile**
asString	**messageText**	**version**

Inherited Protected Data

IBase		
recoverable	**unrecoverable**	

IDate

Derivation

IBase
  IDate

Inherited By

None.

Header File

idate.hpp

Members

Member	Page	Member	Page
Constructor	353	monthOfYear	357
asCDATE	353	operator !=	352
asString	354	operator +	356
dayName	354	operator +=	356
dayOfMonth	354	operator -	356
dayOfWeek	354	operator -=	357
dayOfYear	354	operator <	352
daysInMonth	356	operator <=	352
daysInYear	356	operator ==	352
initialize	358	operator >	352
isLeapYear	357	operator >=	352
isValid	357	today	353
julianDate	353	year	358
monthName	356		

Objects of the IDate class represent specified dates. This class also provides general day and date-handling functions. Externally, dates consist of three pieces of information:

- A year
- A month within that year
- A day within that month

The library also lets you specify the day within the year.

The IDate class returns language-sensitive information, such as names of days and months, in the language defined by the user's system. See the description of the standard C function setlocale in the *VisualAge C++: C Library Reference* for information about setting the locale.

Public Functions

Comparisons

Use these members to compare two IDates. Use any of the full complement of comparison operators and applying the natural meaning.

operator != If the IDate objects represent different dates, true is returned.

```
Boolean operator !=( const IDate& aDate) const;
```

operator < If the left-hand operand represents a date prior to the date represented by the right-hand operand, true is returned.

```
Boolean operator <( const IDate& aDate) const;
```

operator <= If the left-hand operand represents a date prior to or identical to the date represented by the right-hand operand, true is returned.

```
Boolean operator <=( const IDate& aDate) const;
```

operator == If the IDate objects represent the same date, true is returned.

```
Boolean operator ==( const IDate& aDate) const;
```

operator > If the left-hand operand represents a date subsequent to the date represented by the right-hand operand, true is returned.

```
Boolean operator >( const IDate& aDate) const;
```

operator >= If the left-hand operand represents a date subsequent to or identical to the date represented by the right-hand operand, true is returned.

```
Boolean operator >=( const IDate& aDate) const;
```

Constructors

You can construct objects of this class in the following ways:

- Use the default constructor, which returns the current day.
- Give the year, month, and day for the desired day. These parameters can be in either month/day/year or day/month/year order.
- Give the year and day of the year for the desired day.
- Use IDate::today (p. 353) to return the current date.
- Copy another IDate object.
- Give the Julian day number, as a long.
- Give a container details CDATE structure.

Constructors

1 `IDate();`

Using the default constructor returns the current day.

2 `IDate( Month aMonth, int aDay, int aYear);`

You use this constructor when passing parameters in month/day/year order.

3 `IDate( int aDay, Month aMonth, int aYear);`

You use this constructor when passing parameters in day/month/year order.

4 `IDate( int aYear, int aDay);`

This constructor constructs an IDate from the year and day of the year. The day of year is the number of days starting at January 1.

5 `IDate( const IDate& aDate);`

This constructor constructs an IDate by copying another IDate object.

6 `IDate( unsigned long julianDayNumber);`

Use this constructor to construct an IDate from a Julian day number, as a long.

7 `IDate( const _CDATE& cDate);`

Supported On:
PM

You use this constructor to construct an IDate from a container details CDATE structure.

Conversions

Use these members to retrieve other representations of the date.

asCDATE Returns a container CDATE structure for the date.

```
_CDATE asCDATE() const;
```

Supported On:
PM

julianDate Returns the Julian day number of the receiver IDate. This function uses the true definition of a Julian date, which means it returns the number of days from January 1, 4713 B.C.

```
unsigned long julianDate() const;
```

Current Date

Use this member when you need the current date.

today Returns the current date. This static function can be used as an IDate constructor.

```
static IDate today();
```

Day Queries

Use these members to access the day portion of an IDate object.

dayName Returns the name of the receiver's day of the week:

- The first version of dayName accepts a specified day. It returns the name of the day of the week that is equivalent to the index value in *aDay*.
- The second version of dayName accepts no parameters. It returns the name of the receiver's day of the week, such as "Monday".

```
IString dayName() const;
static IString dayName( DayOfWeek aDay);
```

dayOfMonth Returns the day in the receiver's month as an integer from 1 to 31.

```
int dayOfMonth() const;
```

dayOfWeek Returns the index of the receiver's day of the week: Monday through Sunday.

```
DayOfWeek dayOfWeek() const;
```

dayOfYear Returns the day in the receiver's year as an integer from 1 to 366.

```
int dayOfYear() const;
```

Diagnostics

These members provide an IString representation for an IDate object and the capability to output the object to a stream. The formats include both mm-dd-yy and strftime conversion specifications. Often, you use these members to write trace information when debugging.

asString Returns the IDate as a string. The default is formatted per the system (mm-dd-yy). The alternate version of asString lets you use any strftime conversion specifiers. For example, "%x" yields a string such as "Apr 10 1959".

There are two implementations of asString. The parameters are the following:

yearFmt	Specifies how the system will display the year. If you do not specify the format, the default is yy. Use the enumeration IDate::YearFormat for valid *yearFmt* values.
fmt	Specifies the conversion specifier, which is a character string you can use to describe how to output the date. Use the date specifiers that are valid in the C function strftime. The conversion specifiers that apply to IDate and their meanings are listed in the following table. ITime::asString (p. 529) provides the conversion specifiers that apply to ITime. For more information about the strftime function, refer to the *VisualAge C++: C Library Reference*.

Specifier	Meaning
%a	Insert abbreviated weekday name of locale.
%A	Insert full weekday name of locale.
%b	Insert abbreviated month name of locale.
%B	Insert full month name of locale.
%c	Insert date and time of locale.
%d	Insert day of the month (01-31).
%j	Insert day of the year (001-366).
%m	Insert month (01-12).
%U	Insert week number of the year (00-53) where Sunday is the first day of the week.
%w	Insert weekday (0-6) where Sunday is 0.
%W	Insert week number of the year (00-53) where Monday is the first day of the week.
%x	Insert date representation of locale.
%y	Insert year without the century (00-99).
%Y	Insert year.

For example, if you want to return the month, day, and year (without the century), construct an IDate object, and then call asString as follows:

```
asString("%m:%d:%y")
```

```
IString asString( const char* fmt) const;
IString asString( YearFormat yearFmt = yy) const;
```

General Date Queries

These members are static. They provide general IDate utilities independent of specific IDates. Typically, you use them to determine calendar information or to convert IDate enumeration data to string values.

dayName Returns the name of the receiver's day of the week:

- The first version of dayName accepts a specified day. It returns the name of the day of the week that is equivalent to the index value in *aDay*.
- The second version of dayName accepts no parameters. It returns the name of the receiver's day of the week, such as "Monday".

```
static IString dayName( DayOfWeek aDay);
IString dayName() const;
```

daysInMonth Returns the number of days in a specified month of a specified year. You must specify *aYear* in yyyy format.

```
static int daysInMonth( Month aMonth, int aYear);
```

daysInYear Returns the number of days in a specified year. You must specify *aYear* in yyyy format.

```
static int daysInYear( int aYear);
```

monthName Returns the name of the receiver's month:

- The first version of this function accepts no parameters. It returns the name of the receiver's month, such as "March".
- The second version of this function accepts a specified month. It returns the name of the month that is equivalent to the index value in *aMonth*.

```
static IString monthName( Month aMonth);
IString monthName() const;
```

Manipulation

Use these members to update an IDate object using addition or subtraction of another IDate object. Use any of the full complement of addition or subtraction operators and apply the natural meaning.

operator + Adds an integral number of days to the left-hand operand, yielding a new IDate.

```
IDate operator +( int numDays) const;
```

operator += Adds an integral number of days to the left-hand operand, assigning the result to that operand.

```
IDate& operator +=( int numDays);
```

operator - Subtracts an integral number of days from the left-hand operand, yielding a new IDate. If the right-hand operand is also an IDate, the operator yields the number of days between the dates.

The parameters are the following:

numDays	The function subtracts *numDays* from the receiver's value and returns an IDate object.
aDate	The function returns the difference in the number of days between the receiver and *aDate*. If the receiver is greater than *aDate*, the difference is positive.

```
IDate operator -( int numDays) const;
long operator -( const IDate& aDate) const;
```

operator -= Subtracts an integral number of days from the right-hand operand, assigning the result to that operand.

```
IDate& operator -=( int numDays);
```

Month Queries

Use these members to access the month portion of an IDate object.

monthName Returns the name of the receiver's month:

- The first version of this function accepts no parameters. It returns the name of the receiver's month, such as "March".
- The second version of this function accepts a specified month. It returns the name of the month that is equivalent to the index value in *aMonth.*

```
IString monthName() const;
static IString monthName( Month aMonth);
```

monthOfYear Returns the index of the receiver's month of the year: January through December.

```
Month monthOfYear() const;
```

Validation

Use these static members to validate the passed-date data. They test the validity of a given day and provide a leap year test for a given year.

isLeapYear If the specified year is a leap year, true is returned. Otherwise, false is returned. You must specify *aYear* in yyyy format.

```
static Boolean isLeapYear( int aYear);
```

isValid Indicates whether the specified date is valid. You must specify *aYear* in *yyyy* format. You can specify the date as:

- month/day/year
- day/month/year
- year/day

For example, February 29, 1990 is not a valid date because February only had 28 days in 1990.

```
static Boolean isValid( int aDay, Month aMonth, int aYear);
static Boolean isValid( Month aMonth, int aDay, int aYear);
static Boolean isValid( int aYear, int aDay);
```

Year Queries

Use this member to access the year portion of an IDate object.

year Returns the receiver's year. The returned value is in the yyyy format.

```
int year() const;
```

Inherited Public Functions

IBase		
asDebugInfo	**messageFile**	**setMessageFile**
asString	**messageText**	**version**

Protected Functions

Implementation

These members initializes an IDate object.

initialize Calculates the Julian day number. The form of the parameters are the following:

aMonth	mm
aDay	dd
aYear	yyyy

This function returns a reference to the receiver, initialized to the specified date.

```
IDate& initialize( Month aMonth, int aDay, int aYear);
```

Inherited Protected Data

IBase		
recoverable	**unrecoverable**	

Nested Type Definitions

DayOfWeek

```
typedef enum { Monday = 0 , Tuesday , Wednesday , Thursday ,
               Friday , Saturday , Sunday } DayOfWeek;
```

A typedef that provides the values Monday through Sunday for the days of the week.

Month

```
typedef enum { January = 1 , February , March , April ,
               May , June , July , August ,
               September , October , November , December } Month;
```

A typedef that provides the values January through December for the months of the year.

YearFormat

```
typedef enum { yy , yyyy } YearFormat;
```

A typedef that specifies the number of digits in the year for the default asString format (yy or yyyy).

IDBCSBuffer

Derivation

```
IBase
  IVBase
    IBuffer
      IDBCSBuffer
```

Inherited By None.

Header File idbcsbuf.hpp

Members

Member	Page	Member	Page
Constructor	367	isValidMBCS	364
allocate	362	lastIndexOf	365
center	362	lastIndexOfAnyBut	365
charLength	368	lastIndexOfAnyOf	366
charType	364	leftJustify	362
className	368	lowerCase	362
includesDBCS	363	maxCharLength	368
includesMBCS	363	next	364
includesSBCS	363	overlayWith	362
indexOf	364	prevCharLength	368
indexOfAnyBut	364	remove	362
indexOfAnyOf	365	reverse	362
insert	362	rightJustify	363
isCharValid	368	startBackwardsSearch	369
isDBCS	364	startSearch	370
isDBCS1	369	strip	363
isMBCS	364	subString	366
isPrevDBCS	369	translate	363
isSBC	369	upperCase	363
isSBCS	364	~IDBCSBuffer	368
isValidDBCS	364		

Objects of the IDBCSBuffer class implement the version of IString (p. 469) contents that supports mixed double-byte character set (DBCS) characters. This class also supports UNIX multiple-byte character set (MBCS) characters. This class ensures that multiple-byte characters are processed properly.

The use of this class is transparent to the user of class IString.

Public Functions

Allocation

Use these members to reimplement the allocation members as public.

allocate Returns a new buffer of the specified length.

```
IBuffer* allocate( unsigned newLen) const;
```

Editing

Use these members to reimplement the following IString versions of IBuffer members. The following members are called by the corresponding IString members to edit the buffer's contents.

center Centers the receiver within a string of the specified length.

```
IBuffer* center( unsigned newLen, char padCharacter);
```

insert Inserts the specified string after the specified location.

```
IBuffer* insert( const char* pInsert, unsigned insertLen,
    unsigned pos, char padCharacter);
```

leftJustify Left-justifies the receiver in a string of the specified length. If the new length (*length*) is larger than the current length, the string is extended by the pad character (*padCharacter*). The default pad character is a blank.

```
IBuffer* leftJustify( unsigned newLen, char padCharacter);
```

lowerCase Translates all upper-case letters in the receiver to lower-case.

```
IBuffer* lowerCase();
```

overlayWith Replaces a specified portion of the receiver's contents with the specified string. If *pos* is beyond the end of the receiver's data, it is padded with the pad character (*padCharacter*).

```
IBuffer* overlayWith( const char* overlay,
    unsigned len, unsigned pos, char padCharacter);
```

remove Deletes the specified portion of the string (that is, the substring) from the receiver. You can use this function to truncate an IString object at a specific position. For example: `aString.remove(8);` removes the substring beginning at index 8 and takes the rest of the string as a default.

```
IBuffer* remove( unsigned startPos, unsigned numChars);
```

reverse Reverses the receiver's contents.

```
IBuffer* reverse();
```

rightJustify Right-justifies the receiver in a string of the specified length. If the receiver's data is shorter than the requested length (*length*), it is padded on the left with the pad character (*padCharacter*). The default pad character is a blank.

```
IBuffer* rightJustify( unsigned newLen, char padCharacter);
```

strip Strips both leading and trailing character or characters. You can specify the character or characters as the following:

- A char* array
- An IStringTest (p. 515) object

The default is white space.

```
IBuffer* strip( const IStringTest& aTest,
    IStringEnum::StripMode mode);

IBuffer* strip( const char* pChars, unsigned len,
    IStringEnum::StripMode mode);
```

translate Converts all of the receiver's characters that are in the first specified string to the corresponding character in the second specified string.

```
IBuffer* translate( const char* pInputChars,
    unsigned inputLen, const char* pOutputChars,
    unsigned outputLen, char padCharacter);
```

upperCase Translates all lower-case letters in the receiver to upper-case.

```
IBuffer* upperCase();
```

NLS Testing

Use these members to reimplement the following IString versions of IBuffer members. The corresponding IString members use these members to test the buffer's contents. These tests are character set specific.

includesDBCS

If any characters are DBCS (double-byte character set), true is returned.

```
Boolean includesDBCS() const;
```

includesMBCS

If any characters are MBCS (multiple-byte character set), true is returned.

```
Boolean includesMBCS() const;
```

includesSBCS

If any characters are SBCS (single-byte character set), true is returned.

```
Boolean includesSBCS() const;
```

isDBCS If all the characters are DBCS, true is returned.

```
Boolean isDBCS() const;
```

isMBCS If all the characters are MBCS, true is returned.

```
Boolean isMBCS() const;
```

isSBCS If all the characters are SBCS, true is returned.

```
Boolean isSBCS() const;
```

isValidDBCS If no DBCS characters have a 0 second byte, true is returned.

```
Boolean isValidDBCS() const;
```

isValidMBCS If no MBCS characters have a 0 second byte, true is returned.

```
Boolean isValidMBCS() const;
```

Queries

Use these members to reimplement the following IString versions of IBuffer members.

charType Returns the type of a character at the specified index.

```
IStringEnum::CharType charType( unsigned index) const;
```

next Returns a pointer to the next character, not the next byte, in the buffer.

```
const char* next( const char* prev) const;
char* next( const char* prev);
```

Searches

Use these members to reimplement the following IString versions of IBuffer search members.

indexOf Returns the byte index of the first occurrence of the specified string within the receiver. If there are no occurrences, 0 is returned.

```
unsigned indexOf( const IStringTest& aTest,
    unsigned startPos = 1) const;

unsigned indexOf( const char* pString,
    unsigned len, unsigned startPos = 1) const;
```

indexOfAnyBut

Returns the index of the first character of the receiver that is not in the specified set of characters. If there are no characters, 0 is returned. Alternatively, this function returns the index of the first character that fails the test prescribed by a specified IStringTest (p. 515) object.

```
unsigned indexOfAnyBut( const IStringTest& aTest,
    unsigned startPos = 1) const;

unsigned indexOfAnyBut( const char* pString,
    unsigned len, unsigned startPos = 1) const;
```

indexOfAnyOf

Returns the index of the first character of the receiver that is a character in the specified set of characters. If there are no characters, 0 is returned. Alternatively, this function returns the index of the first character that passes the test prescribed by a specified IStringTest (p. 515) object.

```
unsigned indexOfAnyOf( const char* pString,
    unsigned len, unsigned startPos = 1) const;

unsigned indexOfAnyOf( const IStringTest& aTest,
    unsigned startPos = 1) const;
```

lastIndexOf

Returns the index of the last occurrence of the specified string or character. The search starts at the position specified by *startPos* (inclusive) and proceeds backward. The returned value is in the range *0 <= x <= startPos*. The default of 0 starts the search at the end of the receiver's string. If the search target is not found, 0 is returned.

If you specify 0 or 1for *startPos*, this function returns 0 indicating the search target was not found.

```
unsigned lastIndexOf( const char* pString,
    unsigned len, unsigned startPos = 0) const;

unsigned lastIndexOf( const IStringTest& aTest,
    unsigned startPos = 1) const;
```

lastIndexOfAnyBut

Returns the index of the last character not in the specified string or character. The search starts at the position specified by *startPos* (inclusive) and proceeds backward. The default of 0 starts the search at the end of the receiver's string. If the search target is not found, 0 is returned.

If you specify 0 for *startPos*, this function returns 0 indicating the search target was not found.

```
unsigned lastIndexOfAnyBut( const char* pString,
    unsigned len, unsigned startPos = 0) const;

unsigned lastIndexOfAnyBut( const IStringTest& aTest,
    unsigned startPos = 0) const;
```

lastIndexOfAnyOf

Returns the index of the last character in the specified string or character. The search starts at the position specified by *startPos* (inclusive) and proceeds backward. The default of 0 starts the search at the end of the receiver's string. If the search target is not found, 0 is returned.

If you specify 0 or 1 for *startPos*, this function returns 0 indicating the search target was not found.

```
unsigned lastIndexOfAnyOf( const char* pString,
    unsigned len, unsigned startPos = 0) const;

unsigned lastIndexOfAnyOf( const IStringTest& aTest,
    unsigned startPos = 0) const;
```

Subset

Use these members to reimplement the following IString versions of IBuffer subsetting members.

subString

Returns a new IBuffer, of the same type as the previous one, containing the specified subset of characters.

The parameters are the following:

startPos
: The index at which to start the substring. If *startPos* is 0, the function uses position 1. If *startPos* is beyond the end of the buffer, nothing is copied. The buffer is filled out by the specified padding character.

len
: The length to copy from the buffer. If the length extends beyond the end of the buffer, only the portion up to the end is copied. The buffer is then padded. If *len* is 0, a reference to the NULL buffer is returned.

padCharacter
: Specifies the character the function uses to pad the copied string if less than *len* bytes have been copied from the source buffer.

```
IBuffer* subString( unsigned startPos,
    unsigned len, char padCharacter) const;
```

Inherited Public Functions

IBuffer		
addRef	isAlphabetic	lastIndexOfAnyOf
asDebugInfo	isAlphanumeric	leftJustify
center	isASCII	length
change	isControl	lowerCase

IBuffer		
charType	isDBCS	newBuffer
checkAddition	isDigits	next
checkMultiplication	isGraphics	null
compare	isHexDigits	**overflow**
contents	isLowerCase	overlayWith
copy	isMBCS	remove
defaultBuffer	isPrintable	removeRef
fromContents	isPunctuation	reverse
includesDBCS	isSBCS	rightJustify
includesMBCS	isUpperCase	**setDefaultBuffer**
includesSBCS	isValidDBCS	strip
indexOf	isValidMBCS	subString
indexOfAnyBut	isWhiteSpace	translate
indexOfAnyOf	lastIndexOf	upperCase
insert	lastIndexOfAnyBut	useCount

IVBase		
asDebugInfo	asString	

IBase		
asDebugInfo	**messageFile**	**setMessageFile**
asString	**messageText**	**version**

Protected Functions

Constructor and Destructor

The constructor for this class is protected. Only IDBCSBuffer::allocate (p. 362) and IBuffer::initialize (p. 344) can call the constructor.

Constructor

```
IDBCSBuffer( unsigned bufLength);
```

Constructs a buffer of the specified length. The allocated "data" member array actually is 1 byte greater than the argument value (this is achieved automatically via use of the overloaded operator new for class IBuffer). The terminating (extra) byte is set to null.

This constructor is protected. IDBCSBufferss must be obtained by using IDBCSBuffer::nullBuffer and subsequent newBuffer calls to existing IDBCSBuffer objects.

Destructor

```
~IDBCSBuffer();
```

Protected Queries

These members help implement this class.

charLength

Returns the number of bytes in the character whose first byte is pointed to by char *. This is a static function.

```
static size_t charLength( char const*);
size_t charLength( unsigned pos) const;
```

className

Returns the name of the class (IDBCSBuffer).

```
const char* className() const;
```

maxCharLength

Returns the maximum number of bytes in a multiple-byte character. This is a static function.

```
static size_t maxCharLength();
```

prevCharLength

Returns the number of bytes in the preceding character to the one at the specified offset.

```
size_t prevCharLength( unsigned pos) const;
```

Protected Testing

These members help implement this class.

isCharValid

If the character at the specified index is in the set of valid characters, true is returned.

The parameters are the following:

pos
The position in the receiver's buffer for the validity check.

Warning: It is important that this position not be the second byte of a DBCS character. If it is, you might get false results.

pValidChars
The string of the valid characters. It can contain a mixture of DBCS and SBCS characters.

numValidChars
The size of this string of valid characters.

```
Boolean isCharValid( unsigned pos,
    const char* pValidChars, unsigned numValidChars) const;
```

isDBCS1
If the byte at the specified offset is the first byte of DBCS, true is returned.

Note: The lxbrary provides this function only for compatibility with prior library versions. We recommend using IDBCSBuffer::charLength (p. 368) to determine if the byte is part of a multiple-byte character.

```
Boolean isDBCS1( unsigned pos) const;
```

Supported On:
PM

isPrevDBCS
If the preceding character to the one at the specified offset is a DBCS character, true is returned.

Note: The library provides this function only for compatibility with prior library versions. We recommend using IDBCSBuffer::prevCharLength (p. 368) to determine if the preceding byte is part of a multiple-byte character.

```
Boolean isPrevDBCS( unsigned pos) const;
```

Supported On:
PM

isSBC
If the byte pointed to by the specified character is a single-byte character, true is returned. This is a static function.

```
static Boolean isSBC( char const*);
```

Search Initialization

These members help implement this class. They initialize search data.

startBackwardsSearch

Initializes a search of type IString::lastIndexOf (p. 489).

- If *searchLen* is greater than the length of the buffer, 0 is returned indicating an invalid search request.
- If the starting position is 0 or beyond the last *searchLen* bytes of the buffer, the position where the last *searchLen* bytes start in the buffer is returned.
- If the starting position is 1 through the last *searchLen* bytes, the value of *startingPos* is returned.

```
unsigned startBackwardsSearch( unsigned startPos,
    unsigned searchLen) const;
```

startSearch Initializes a search of type IString::indexOf (p. 481).

- If *startPos* is 0, the search uses a starting position of 1.
- If the specified *startPos* and *searchLen* result in an invalid search, 0 is returned. This usually occurs when the sum of *startPos* and *searchLen* is greater than the size of the buffer.

```
unsigned startSearch( unsigned startPos, unsigned searchLen) const;
```

Inherited Protected Functions

IBuffer		
allocate	**initialize**	operator new
className	operator delete	startBackwardsSearch

Inherited Public Data

IBuffer		
dbcsTable		

Inherited Protected Data

IBase		
recoverable	**unrecoverable**	

IDeviceError

Derivation

IException
  IDeviceError

Inherited By None.

Header File iexcbase.hpp

Members

Member	Page
Constructor	371
name	372

Objects of the IDeviceError class represent an exception. When a member function makes a hardware-related request of the operating system or the presentation system that the system cannot satisfy because of a hardware failure, the member function creates and throws an object of the IDeviceError class. An example of a failing hardware-related request is printing to a disconnected printer.

Public Functions

Constructor

You can construct objects of this class.

Constructor You can create objects of this class by doing the following:

- Using the constructor.

  errorText The text describing this particular error.

  errorId The identifier you want to associate with this particular error.

  severity Use the enumeration IException::Severity (p. 386) to specify the severity of the error. The default is unrecoverable.

- Using the macros discussed in IException (p. 379). The library provides these macros to make creating exceptions easier for you.

```
IDeviceError( const char* errorText, unsigned long errorId,
   Severity severity = IException::unrecoverable);
```

Exception Type

These members provide support for determining the name (type) of the exception. This is used for logging out an exception object's error information.

name Returns the name of the object’s class.

```
virtual const char* name() const;
```

Inherited Public Functions

IException		
addLocation	locationAtIndex	setSeverity
appendText	locationCount	setText
assertParameter	logExceptionData	**setTraceFunction**
errorCodeGroup	name	terminate
errorId	setErrorCodeGroup	text
isRecoverable	setErrorId	textCount

Inherited Public Data

IException		
baseLibrary	**CLibrary**	**operatingSystem**

IErrorInfo

Derivation

IBase
  IVBase
    IErrorInfo

Inherited By

ICLibErrorInfo
IGUIErrorInfo
IMMErrorInfo
ISystemErrorInfo
IXLibErrorInfo

Header File

iexcept.hpp

Members

Member	Page	Member	Page
Constructor	375	text	376
errorId	375	throwError	376
isAvailable	376	~IErrorInfo	375
operator const char *	376		

The IErrorInfo class is an abstract base class that defines the interface for its derived classes. These classes retrieve error information and text that you can subsequently use to create an exception object. The following macros assist in throwing exceptions:

IASSERTPARM
This macro accepts an expression to test. The expression is asserted to be true. If it evaluates to false, the macro generates code that calls the IExcept__assertParameter function, which creates an IInvalidParameter (p. 395) exception. The error group, other, is added to the object. The exception data is logged using IException::TraceFn::logExceptionData, and the exception is then thrown.

IASSERTSTATE
This macro accepts an expression to test. The expression is asserted to be true. If it evaluates to false, the macro generates code that calls the IExcept__assertState function, which creates an IInvalidRequest (p. 397) exception. The error group, other, is added to the object. The exception data is logged, and the exception is then thrown.

ITHROWLIBRARYERROR
This macro can throw any of the library-defined exceptions.

id	The ID of the message to load from the class library message file.

name A value from the enumeration IErrorInfo::ExceptionType (p. 377), indicating the type of exception to create.

severity A value from the enumeration IException::Severity (p. 386), indicating the severity of the exception.

The macro generates code that calls the IExcept__throwLibraryError function, which does the following:

1. Loads the message text from the class library message file
2. Uses the message text to create an exception object
3. Adds location information
4. Logs the exception data
5. Throws the exception

ITHROWLIBRARYERROR1

This macro can throw any of the library-defined exceptions. It is identical to the ITHROWLIBRARYERROR macro, except it has a fourth parameter:

text Replacement text for the retrieved message.

ITHROWERROR

This macro can throw any of the library-defined exceptions.

messageId The ID of the message to load from the message file.

name A value from the enumeration IErrorInfo::ExceptionType (p. 377), indicating the type of exception to create.

severity A value from the enumeration IException::Severity (p. 386), indicating the severity of the exception.

messageFile
The name of the message file to load the exception text from. This name should include the file extension. e.g. "USERMSG.MSG"

errorGroup
The errorGroup associated with this error. This can be one of the values for ErrorCodeGroup defined in IException, or a value you provide.

The macro generates code that calls the IExcept__throwError function, which does the following:

1. Loads the message text from the specified library message file
2. Uses the message text to create an exception object
3. Adds the error group to the object

4. Adds location information
5. Logs the exception data
6. Throws the exception

ITHROWERROR1
This macro can throw any of the library-defined exceptions. It is identical to the ITHROWERROR macro, except it has a fourth parameter:

substitutionText
Substitution text for the retrieved message.

PM IGUIErrorInfo (p. 391), ISystemErrorInfo (p. 523), and ICLibErrorInfo (p. 347) are derived from this class. You can use IGUIErrorInfo to obtain information about errors detected by the Win calls for Presentation Manager. Use ISystemErrorInfo to obtain error information about DOS system call errors.

Motif IXLibErrorInfo (p. 543) is derived from this class. You can use IXLibErrorInfo to obtain error information about error conditions detected when calling X library APIs. Use ICLibErrorInfo to obtain error information about error conditions detected when calling C Library functions.

You can create objects of IGUIErrorInfo (p. 391) and ISystemErrorInfo (p. 523) on AIX, but they have default messages:

IGUIErrorInfo	GUI exception condition detected
ISystemErrorInfo	System exception condition detected

Public Functions

Constructor and Destructor

This is a virtual base class so you cannot create objects of this type without deriving from this class.

Constructor
```
IErrorInfo();
```

Destructor
```
virtual ~IErrorInfo();
```

Error Information

Use these members to return error information provided by objects of this class. All the members are pure virtual.

errorId Returns the error ID.
```
virtual unsigned long errorId() const = 0;
```

isAvailable

If error information is available, true is returned.

```
virtual Boolean isAvailable() const = 0;
```

operator const char *

Returns the error text.

```
virtual operator const char *() const = 0;
```

text

Returns the error text.

```
virtual const char* text() const = 0;
```

Throw Support

Use these members to support the throwing of exceptions.

throwError

Creates an IErrorInfo object and uses it to do the following:

1. Create an exception object
2. Add the error code group to the object
3. Add the location information to the object
4. Log the exception data
5. Throw the exception

location An IExceptionLocation (p. 389) object containing the following:

- Function name
- File name
- Line number where the function is called

name Use the enumeration ExceptionType (p. 377) to specify the type of the exception. The default is accessError.

severity Use the enumeration IException::Severity (p. 386) to specify the severity of the error. The default is recoverable.

errorGroup
Use one of the ErrorCodeGroup values provided in IException, or provide your own group for this parameter. The default is baseLibrary.

```
void throwError( const IExceptionLocation& location,
    ExceptionType name = accessError,
    IException::Severity severity = recoverable,
    IException::ErrorCodeGroup errorGroup = IException::baseLibrary);
```

Inherited Public Functions

IVBase		
asDebugInfo	asString	

IBase		
asDebugInfo	**messageFile**	**setMessageFile**
asString	**messageText**	**version**

Inherited Protected Data

IBase		
recoverable	**unrecoverable**	

ExceptionType

```
ExceptionType {
    accessError,    deviceError,         invalidParameter,
    invalidRequest, outOfSystemResource, outOfWindowResource,
    outOfMemory,    resourceExhausted
    };
```

The following enumeration type is defined to specify the type of exception to create on various functions and macros:

ExceptionType - Used to specify the type of exception to be created. The allowable values are:

accessError
Creates an IAccessError object.

deviceError
Creates an IDeviceError object.

invalidParameter
Creates an IInvalidParameter object.

invalidRequest
Creates an IInvalidRequest object.

outOfSystemResource
Creates an IOutOfSystemResource object.

outOfWindowResource
Creates an IOutOfWindowResource object.

outOfMemory
Creates an IOutOfMemory object.

resourceExhausted
Creates an IResourceExhausted object.

IException

Derivation Inherits from none.

Inherited By

IAccessError	IInvalidParameter
IAssertionFailure	IInvalidRequest
IDeviceError	IResourceExhausted

Header File iexcbase.hpp

Members

Member	Page	Member	Page
Constructor	382	operatingSystem	386
addLocation	383	other	386
appendText	384	presentationSystem	386
assertParameter	385	setErrorCodeGroup	382
baseLibrary	386	setErrorId	383
CLibrary	386	setSeverity	384
errorCodeGroup	382	setText	384
errorId	383	setTraceFunction	384
isRecoverable	384	terminate	381
locationAtIndex	383	text	385
locationCount	383	textCount	385
logExceptionData	384	~IException	382
name	385		

The IException class is the base class from which all exception objects thrown in the library are derived. None of the functions in this class throws exceptions because an exception has probably already been thrown or is about to be thrown. Member functions in the library create objects of classes derived from IException for all error conditions the functions encounter. Each exception object contains the following:

- A stack of exception message text strings (descriptions)
- An error ID
- A severity code
- An error code group
- Information about where the exception was thrown

IException provides all of the functions required for it and its derived classes, including functions that operate on the text strings in the stack.

IException

The library defines the derived classes so that you can catch exceptions by their type. In general, never create an IException object. Instead, create and throw an object of the appropriate derived class. The derived classes of IException are the following:

IAccessError (p. 319)
IAssertionFailure (p. 321)
IDeviceError (p. 371)
IInvalidParameter (p. 395)
IInvalidRequest (p. 397)
IResourceExhausted (p. 459)

In addition, IResourceExhausted has the following derived classes:

IOutOfMemory (p. 419)
IOutOfSystemResource (p. 421)
IOutOfWindowResource (p. 423)

You can also derive your own exception type from IException.

The library provides the following macros to assist in using exception handling. If you derive your own exception type and you want to use a macro, you must use the ITHROW macro or write your own macro.

ITHROW
Accepts as input a predefined object of any IException-derived class. The macro generates code to add the location information to the objects, logs all object data, and throws the exception.

IRETHROW
Accepts as input an object of any derived class of IException that has been previously thrown and caught. Like the ITHROW macro, it also captures the location information and logs all object data before re-throwing the exception.

IASSERT
If you define IC_DEVELOP during the compile for debugging purposes, this macro expands to provide assertion support in the library. This macro accepts an expression to test. If the test evaluation returns false, IASSERT calls assertParameter (p. 385).

IEXCLASSDECLARE
Creates a declaration for a derived class of IException or one of its derived classes.

IEXCLASSIMPLEMENT
Creates a definition for a derived class of IException or one of its derived classes.

IEXCEPTION_LOCATION
Expands to create an object of the class IExceptionLocation (p. 389).

INO_EXCEPTIONS_SUPPORT
Supports compilers lacking an exception-handling implementation. If you use the INO_EXCEPTIONS_SUPPORT macro, the following macros end the program after capturing the location information and logging it. These macros normally throw an exception.

ITHROW	Found in IException.
IASSERTPARM	Found in IErrorInfo (p. 373).
IASSERTSTATE	Found in IErrorInfo.
ITHROWERROR	Found in IErrorInfo.
ITHROWERROR1	Found in IErrorInfo.
ITHROWLIBRARYERROR	Found in IErrorInfo.
ITHROWLIBRARYERROR1	Found in IErrorInfo.
ITHROWGUIERROR	Found in IGUIErrorInfo (p. 391).
ITHROWGUIERROR2	Found in IGUIErrorInfo.
ITHROWSYSTEMERROR	Found in ISystemErrorInfo (p. 523).

Warning: The INO_EXCEPTIONS_SUPPORT macro might not work correctly on all compilers.

Whenever the library throws one of these exceptions, trace records are output with information about the exception. The class ITrace (p. 533) describes tracing in more detail.

Public Functions

Application Termination

These members provide support for terminating an application instead of throwing an exception.

terminate Ends the application. Normally, the library only intends this function to be used internally by the library's exception handling macros when the compiler you are using does not support C++ exception handling. This only occurs if you define the INO_EXCEPTIONS_SUPPORT macro. The macros that use this function are:

ITHROW	Found in IException.
IASSERTPARM	Found in IErrorInfo (p. 373).
IASSERTSTATE	Found in IErrorInfo.
ITHROWLIBRARYERROR	Found in IErrorInfo.
ITHROWLIBRARYERROR1	Found in IErrorInfo
ITHROWGUIERROR	Found in IGUIErrorInfo (p. 391).
ITHROWGUIERROR2	Found in IGUIErrorInfo.
ITHROWSYSTEMERROR	Found in ISystemErrorInfo (p. 523).

```
virtual void terminate();
```

Constructors and Destructor

You can construct and destruct objects of this class. You cannot assign one IException object from another.

Constructors

1

```
IException( const char* errorText,
    unsigned long errorId = 0,
    Severity severity = IException::unrecoverable);
```

You can construct objects of this class by doing the following:

- Using the primary constructor. Normally, this is the only way you can construct an object of this class.

errorText	The text describing this error.
errorId	(Optional) The identifier you want to associate with this particular error.
severity	(Optional) Use the enumeration IException::Severity (p. 386) to specify the severity of the error. The default is unrecoverable.

- Using the copy constructor. The library provides this constructor so the compiler can copy the exception when it is thrown.

exception	The exception object you want to copy.

2

```
IException( const IException& exception);
```

The copy constructor is provided so that the compiler can make copies of the object during the throwing of an exception.

Destructor

```
virtual ~IException();
```

Error Code

Use these members to determine which class library an exception originated from.

errorCodeGroup

Returns the error group the exception originated from.

```
ErrorCodeGroup errorCodeGroup() const;
```

setErrorCodeGroup

Sets the id of the originating class library into the exception object.

```
IException& setErrorCodeGroup( ErrorCodeGroup errorGroup);
```

Error Information

Use these members to get or modify the error identifier of the exception object.

errorId

Returns the error ID of the exception.

```
unsigned long errorId() const;
```

setErrorId

Sets the error ID to the specified value.

errorId The identifier you want to associate with this error.

```
IException& setErrorId( unsigned long errorId);
```

Exception Location

Use these members to set and access the location information in the exception object.

addLocation

Adds the location information to the exception object. The library captures this information when an exception is thrown or re-thrown. An array of IExceptionLocation objects is stored in the exception object.

location An IExceptionLocation (p. 389) object containing the following:

- Function name
- File name
- Line number where the function is called

```
virtual IException&
    addLocation( const IExceptionLocation& location);
```

locationAtIndex

Returns the IExceptionLocation (p. 389) object at the specified index.

locationIndex
If the index is not valid, a 0 pointer is returned.

```
const IExceptionLocation*
    locationAtIndex( unsigned long locationIndex) const;
```

locationCount

Returns the number of locations stored in the exception location array.

```
unsigned long locationCount() const;
```

Exception Logging

Use these members to log exception information.

logExceptionData

Logs the exception data stored in the IException object using the function specified by IException::setTraceFunction (p. 384). If you have not set a tracing function, the exception information is written to standard error output.

```
virtual IException& logExceptionData();
```

setTraceFunction

Registers an object of IException::TraceFn (p. 387) to be used to log exception data. The ITrace (p. 533) member functions and macros write the trace messages. IException::logExceptionData (p. 384) calls IException::TraceFn::write (p. 388) during exception processing to write the data. If you do not register an object, data is written to standard error output.

traceFunction
Your own trace function implementation.

```
static void
    setTraceFunction( IException::TraceFn& traceFunction);
```

Exception Severity

Use these members to set and determine the severity of the error condition.

isRecoverable

If the thrower (that is, whatever creates the exception) determines the exception is recoverable, 1 is returned. If the thrower determines it is unrecoverable, 0 is returned.

```
virtual int isRecoverable() const;
```

setSeverity

Sets the severity of the exception.

severity Use the enumeration Severity (p. 386) to specify the severity of the exception.

```
IException& setSeverity( Severity severity);
```

Exception Text

Use these members to set, modify, and retrieve the exception text in the object.

appendText

Appends the specified text to the text string on the top of the exception text stack.

errorText The text you want to append.

```
IException& appendText( const char* errorText);
```

setText

Adds the specified text to the top of the exception text stack.

errorText The error text you want to add.

```
IException& setText( const char* errorText);
```

text Returns a constant char* pointing to an exception text string from the exception text stack.

indexFromTop
The default index is 0, which is the top of the stack. If you specify an index which is not valid, a 0 pointer is returned.

```
const char* text( unsigned long indexFromTop = 0) const;
```

textCount Returns the number of text strings in the exception text stack.

```
unsigned long textCount() const;
```

Exception Type

Use these members to determine the name (type) of the exception. This is used for logging out an exception object's error information.

name Returns the name of the object's class.

```
virtual const char* name() const;
```

Throw Support

These members support the throwing of exceptions.

assertParameter

The IASSERT macro uses this function to do the following:

1. Create an IAssertionFailure (p. 321) exception
2. Add the location information to it
3. Log the exception data
4. Throw the exception

exceptionText
The text describing the exception.

location An IExceptionLocation (p. 389) object containing the following:

- Function name
- File name
- Line number where the function is called

```
static void assertParameter( const char* exceptionText,
    IExceptionLocation location);
```

Public Data

Error Code

Use these members to determine which class library an exception originated from.

baseLibrary
This is the error group for IBM Open Class Library errors.

```
static ErrorCodeGroup const baseLibrary;
```

CLibrary
This is the error group for the C library errors.

```
static ErrorCodeGroup const CLibrary;
```

operatingSystem
This is the error group for operating system errors.

```
static ErrorCodeGroup const operatingSystem;
```

other
This is the error group for errors which don't fall in any of the other groups.

```
static ErrorCodeGroup const other;
```

presentationSystem
This is the error group for presentation system errors.

```
static ErrorCodeGroup const presentationSystem;
```

Nested Classes

IException contains the following nested classes:

IException::TraceFn (see page 387)

Nested Type Definitions

Severity

```
Severity { unrecoverable, recoverable };
```

Use these enumerators to specify the severity of the exception:

unrecoverable
Classifies the exception as unrecoverable.

recoverable
Classifies the exception as recoverable.

ErrorCodeGroup

```
typedef const char * ErrorCodeGroup;
```

This identifies the source of the exception's error code.

IException::TraceFn

Derivation Inherits from none.

Inherited By None.

Header File iexcbase.hpp

Members

Member	Page	Member	Page
Constructor	388	TraceFn	388
exceptionLogged	388	write	388
logData	388		

Objects of the class IException (p. 379) and its derived classes use IException::TraceFn to log exception object data.

A default TraceFn derived object is registered by the Collection Class Library. If the User Interface Library is used, it registers a TraceFn derived object which overrides the write function. It uses ITrace to write out the buffers of data, so the buffers will be written to wherever the ICLUI TRACETO environment variable directs the output from ITrace (p. 533).

If you want to provide your own tracing function, derive your own class from IException::TraceFn and register it with IException using IException::setTraceFunction (p. 384). You can completely take over exception logging by overriding the logData function. You are passed the IException object so you can completely customize the logging of exception data. If you only wish to change how the buffers of exception data are logged you should override the write function.

The exceptionLogged function is provided so that you can determine when the last buffer of exception data has been passed to the write function by the default logData function. This allows you to gather all of the exception data by only overriding the write and exceptionLogged functions for situations where you must write all of the exception data out with one call.

Public Functions

Tracing

The IException's logExceptionData member uses these members to log instance data of exception objects.

logData Logs error information contained in an Exception object.

```
virtual void logData( IException& exception);
```

write Writes a buffer of exception data.

```
virtual void write( const char* buffer);
```

Protected Functions

Constructors

The only way to create objects of this class is from a derived class. To enforce this, the only constructors we provide for this class are protected.

Derived classes use these members to create objects of this class.

Constructors This default constructor can be used by derived classes to create objects of this class.

```
TraceFn();
```

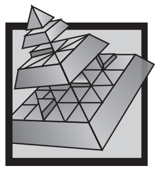
Tracing

The function IException::logExceptionData uses these members to log instance data of exception objects.

exceptionLogged

This function is called by the default logData function after the last buffer of exception data has been passed to the write function.

```
virtual void exceptionLogged();
```



IExceptionLocation

Derivation Inherits from none.

Inherited By None.

Header File iexcbase.hpp

Members

Member	Page	Member	Page
Constructor	390	functionName	389
fileName	389	lineNumber	389

Objects of the IExceptionLocation class save the location information when an exception is thrown or re-thrown. None of the functions in this class throws exceptions because an exception probably has been thrown already or is about to be thrown.

Typically, either the ITHROW or IRETHROW macro creates an IExceptionLocation object when an exception is to be thrown or re-thrown, respectively. However, you can create your own IExceptionLocation object by constructing it yourself or by using the IEXCEPTION_LOCATION macro.

Public Functions

Attributes

Use these members to return the attributes of the exception location object.

fileName Returns the path-qualified source file name where an exception has been thrown or re-thrown.

```
const char* fileName() const;
```

functionName

Returns the name of the function that has thrown or re-thrown an exception.

```
const char* functionName() const;
```

lineNumber Returns the line number of the statement in the source file from which an exception has been thrown or re-thrown.

```
unsigned long lineNumber() const;
```

Constructor

Constructor You can create objects of this class by doing the following:

- Using the constructor.

  fileName The source file containing the function that created this object.

  functionName
  The name of the function creating this object.

  lineNumber
  The line number of the statement from the source file from which the object was created.

- Using the macro IEXCEPTION_LOCATION (p. 379). This macro captures the current location information using constants provided by the compiler for all of the parameters. Default values are provided for all the parameters to support environments in which all constants or alternative means for getting location information are not provided.

```
IExceptionLocation( const char* fileName = 0,
    const char* functionName = 0,
    unsigned long lineNumber = 0);
```

IGUIErrorInfo

Derivation

IBase
  IVBase
    IErrorInfo
      IGUIErrorInfo

Inherited By None.

Header File iexcept.hpp

Members

Member	Page	Member	Page
Constructor	392	text	393
errorId	393	throwGUIError	393
isAvailable	393	~IGUIErrorInfo	393
operator const char *	393		

Objects of the IGUIErrorInfo class represent error information that you can include in an exception object. When an OS/2 Win call results in an error condition, objects of the IGUIErrorInfo class are created. You can use the error text to construct a derived class object of IException (p. 379).

The library provides the following macros for throwing exceptions constructed with IGUIErrorInfo information:

ITHROWGUIERROR
This macro accepts as its only parameter the name of the GUI function that returned an error condition. This macro then generates code that calls IGUIError::throwGUIError (p. 393), which does the following:

1. Creates an IGUIErrorInfo object
2. Uses the object to create an object of IAccessError (p. 319)
3. Adds the presentationSystem error group to the object
4. Adds location information
5. Logs the exception data
6. Throws the exception

Note: This macro uses the recoverable enumerator provided by IException::Severity (p. 386).

ITHROWGUIERROR2

This macro can throw any of the User Interface Class Library-defined exceptions. This macro accepts the following parameters:

location	The name of the GUI function returning an error code, the name of the file the function is in, and the function's line number.
name	Use the enumeration IErrorInfo::ExceptionType (p. 377) to specify the type of the exception. The default is accessError.
severity	Use the enumeration IException::Severity (p. 386) to specify the severity of the error. The default is recoverable.

This macro generates code that calls throwGUIError (p. 393), which does the following:

1. Creates an IGUIErrorInfo object
2. Uses the object to create an IException object
3. Adds the presentationSystem error group to the object
4. Adds location information
5. Logs the exception data
6. Throws the exception

PM You can use objects of the IGUIErrorInfo class to obtain information about the last error that occurred on a call to Presentation Manager.

Motif You can create objects of this class on AIX, but the objects contain no useful information and only have the default message: "GUI exception condition detected."

You can use this class in OS/2 to create error information for GUI errors resulting from Win calls. Objects of this class obtain the error information by calling WinGetLastError, which is the Presentation Manager API that maintains the error information per thread. Motif does not have a similar mechanism where you can query the X server for error information. If you use objects of this class in AIX, they obtain a default message, which is "GUI exception condition detected."

Public Functions

Constructor and Destructor

You can construct and destruct objects of this class. You cannot copy or assign objects of this class.

Constructor

```
IGUIErrorInfo( const char* GUIFunctionName = 0);
```

You can only construct objects of this class using the default constructor.

Note: If the constructor cannot load the error text, the library provides the following default text: "No error text is available."

GUIFunctionName
The name of the failing GUI function. If you specify *GUIFunctionName*, the constructor prefixes it to the error text. Optional.

Destructor

```
virtual ~IGUIErrorInfo();
```

Error Information

Use these members to return error information provided by objects of this class.

errorId

Returns the error ID.

```
virtual unsigned long errorId() const;
```

PM In the case of a Presentation Manager error, the IGUIErrorInfo constructor obtains the *errorId* using WinGetLastError.

isAvailable

If the error information is available, true is returned.

```
virtual Boolean isAvailable() const;
```

operator const char *

Returns the error text.

```
virtual operator const char *() const;
```

text

Returns the error text.

```
virtual const char* text() const;
```

Throw Support

Use these members to support the throwing of exceptions using information from an IGUIErrorInfo object. The throwGUIError function is used by the ITHROWGUIERROR macro.

throwGUIError

Creates an IGUIErrorInfo object and uses the text from it to do the following:

1. Create an exception object
2. Add the location information to it
3. Log the exception data
4. Throw the exception.

functionName
The name of the function where the exception occurred.

location	An IExceptionLocation (p. 389) object containing the following: • Function name • File name • Line number where the function is called
name	Use the enumeration IErrorInfo::ExceptionType (p. 377) to specify the type of the exception. The default is accessError.
severity	Use the enumeration IException::Severity (p. 386) to specify the severity of the error. The default is recoverable.

```
static void throwGUIError( const char* functionName,
    const IExceptionLocation& location,
    IErrorInfo::ExceptionType name = accessError,
    IException::Severity severity = recoverable);
```

Inherited Public Functions

IErrorInfo		
errorId	isAvailable	operator const char *

IVBase		
asDebugInfo	asString	

IBase		
asDebugInfo	**messageFile**	**setMessageFile**
asString	**messageText**	**version**

Inherited Protected Data

IBase		
recoverable	**unrecoverable**	

IInvalidParameter

Derivation

IException
  IInvalidParameter

Inherited By None.

Header File iexcbase.hpp

Members

Member	Page
Constructor	395
name	396

Objects of the IInvalidParameter class represent an exception. When a member function detects an invalid input parameter, the member function creates and throws an object of the IInvalidParameter class. This exception is identical to the exception IAssertionFailure (p. 321), with one difference: IInvalidParameter is thrown whether or not you define IC_DEVELOP for the compile.

Public Functions

Constructor

You can construct objects of this class.

Constructor You can create objects of this class by doing the following:

- Using the constructor.

  errorText The text describing this particular error.

  errorId The identifier you want to associate with this particular error.

  severity Use the enumeration IException::Severity (p. 386) to specify the severity of the error. The default is unrecoverable.

- Using the macros discussed in IException (p. 379). The library provides these macros to make creating exceptions easier for you.

```
IInvalidParameter( const char* errorText, unsigned long errorId,
   Severity severity = IException::unrecoverable);
```

Exception Type

Use these members to determine the name (type) of the exception. This is used for logging out an exception object's error information.

name Returns the name of the object’s class.

```
virtual const char* name() const;
```

Inherited Public Functions

IException		
addLocation	locationAtIndex	setSeverity
appendText	locationCount	setText
assertParameter	logExceptionData	**setTraceFunction**
errorCodeGroup	name	terminate
errorId	setErrorCodeGroup	text
isRecoverable	setErrorId	textCount

Inherited Public Data

IException		
baseLibrary	**CLibrary**	**operatingSystem**

IInvalidRequest

Derivation

IException
  IInvalidRequest

Inherited By None.

Header File iexcbase.hpp

Members

Member	Page
Constructor	397
name	398

Objects of the IInvalidRequest class represent an exception. Whenever an object cannot satisfy a request, the member function creates and throws an object of the IInvalidRequest class. An example of such a request occurs if you try to paste text from the system clipboard, but the clipboard has no data.

Public Functions

Constructor

You can construct objects of this class.

IInvalidRequest

You can create objects of this class by doing the following:

- Using the constructor.

  errorText The text describing this particular error.

  errorId The identifier you want to associate with this particular error.

  severity Use the enumeration IException::Severity (p. 386) to specify the severity of the error. The default is unrecoverable.

- Using the macros discussed in IException (p. 379). The library provides these macros to make creating exceptions easier for you.

```
IInvalidRequest( const char* errorText, unsigned long errorId,
   Severity severity = IException::unrecoverable);
```

Exception Type

Use these members to determine the name (type) of the exception. This is used for logging out an exception object's error information.

name Returns the name of the object’s class.

```
virtual const char* name() const;
```

Inherited Public Functions

IException		
addLocation	locationAtIndex	setSeverity
appendText	locationCount	setText
assertParameter	logExceptionData	**setTraceFunction**
errorCodeGroup	name	terminate
errorId	setErrorCodeGroup	text
isRecoverable	setErrorId	textCount

Inherited Public Data

IException		
baseLibrary	**CLibrary**	**operatingSystem**

IMessageText

Derivation Inherits from none.

Inherited By None.

Header File imsgtext.hpp

Members

Member	Page	Member	Page
Constructor	400	setDefaultText	401
operator =	401	text	401
operator const char *	401	~IMessageText	401

Objects of the IMessageText class load message text from a message file. When the library detects an error condition and prepares to throw an exception, the library creates an object of this class if it is using customized message text. You can use the message text provided by this class to construct an object of a class derived from IException (p. 379).

PM The IMessageText object searches for the message file as follows:

- The system root directory
- The current working directory
- The DPATH environment setting
- The APPEND environment setting

Typically, message files have the extension .MSG.

Motif The IMessageText object searches for the message file using the NLSPATH environment setting.

Public Functions

Constructors and Destructor

You can construct, destruct, copy, and assign objects of this class.

Constructor

1
```
IMessageText( unsigned long messageId,
    const char* messageFileName, const char* textInsert1 = 0,
    const char* textInsert2 = 0, const char* textInsert3 = 0,
    const char* textInsert4 = 0, const char* textInsert5 = 0,
    const char* textInsert6 = 0, const char* textInsert7 = 0,
    const char* textInsert8 = 0, const char* textInsert9 = 0);
```

You can construct objects of this class using this constructor, allowing you to retrieve a message from a file and, optionally, insert additional text strings within the retrieved message.

You can specify that the object insert the text strings through substitution symbols within the message. For example:

```
The application cannot find the file, %1, at the specified path, %2.
```

Using this constructor, you can replace the substitution symbols by supplying the file name and path name via *textInsert1* and *textInsert2* respectively. Notice the substitution symbol number (%1) matches the parameter number (*textInsert1*).

Warning: You must use the numbers in sequence. For example, you cannot use %1, %2, and %5 in a message, skipping %3 and %4. Instead, you must use %1, %2, and %3. You must specify the substitution symbols sequentially and the text insertion parameters' numbers must match their respective substitution symbol.

messageId The message ID.

messageFileName

The name of the message file to retrieve the message from. The message file name must include the file extension.

If you specify 0, the message text is in a message segment bound to the .EXE. The IMessageText object loads the message from the application. Otherwise, the library searches for the message text in the specified message file.

Note: If the library cannot load the text from the message file, this constructor uses the following default text: "Unable to load text from message file."

textInsert1 **through** *textInsert9*
(Optional) A text string you insert into the message.

2
```
IMessageText( const IMessageText& text);
```

You can construct objects of this class using the library provided copy constructor.

text The error message text.

operator = Sets the object data to the values of the specified IMessageText object.

text The message text object you want to copy.

```
IMessageText& operator =( const IMessageText& text);
```

Destructor
```
~IMessageText();
```

Text Operations

Use these members to obtain the text from the object and to set the default text for the object.

operator const char * Returns the message text.

```
operator const char *() const;
```

setDefaultText

Sets the default message text to the specified text string. The text is set only if the constructor cannot load the text for the specified message ID.

Note: The default text is: "Unable to load text from message file."

text The new default text string.

```
IMessageText& setDefaultText( const char* text);
```

text Returns the message text.

```
const char* text() const;
```

INotificationEvent

Derivation

IBase
  INotificationEvent

Inherited By None.

Header File inotifev.hpp

Members

Member	Page	Member	Page
Constructor	403	operator =	404
eventData	404	setEventData	404
hasNotifierAttrChanged	404	setNotifierAttrChanged	405
notificationId	404	setObserverData	405
notifier	404	~INotificationEvent	404
observerData	404		

The class INotificationEvent provides the details of a notification event to an observer object. INotifier objects create notification events when these objects change or when they must notify observer objects of events. All IBM User Interface Class Library classes may inherit from the INotifier class to obtain the ability to notify. Currently, the IBM User Interface Class Library has implemented the IWindow (Vol. II) class as inheriting from INotifier. Therefore, all classes derived from IWindow (Vol. II) inherit this ability.

Public Functions

Constructors and Destructor

You can construct, destruct, and assign objects of this class.

Constructors

1
```
INotificationEvent( const INotificationId& identifier,
    INotifier& notifier,
    Boolean notifierAttrChanged = true,
    const IEventData& eventData = IEventData ( ),
    const IEventData& observerData = IEventData ( ));
```

You can construct an INotificationEvent object using a notification identifier, a reference to a notifier object derived from INotifier, and a Boolean indicator of whether this event describes a change in an attribute of the notifier. The notifier can also include data specific to the particular notification. This data is documented with the notification IDs in the definition of the derived notifier class. The notifier must also add observer data to the event if the observer provided this data when registering with the notifier.

2
```
INotificationEvent( const INotificationEvent& event);
```

You can construct an INotificationEvent object using a copy of an existing notification event.

operator = Replaces the contents of one INotificationId object with another INotification object.

```
INotificationEvent& operator =( const INotificationEvent& event);
```

Destructor

```
~INotificationEvent();
```

Event Attributes

Use these members to get and set the attributes of objects of this class.

eventData Returns the data specific to the event.

```
IEventData eventData() const;
```

hasNotifierAttrChanged

Returns true if the event represents a change in an attribute of the notifier object.

```
Boolean hasNotifierAttrChanged() const;
```

notificationId Returns the INotificationId for the event. The derived INotifier classes document the notification identifiers.

```
INotificationId notificationId() const;
```

notifier Returns a reference to the notifier object.

```
INotifier& notifier() const;
```

observerData Returns observer data that is added when the observer registers with the notifier object.

```
IEventData observerData() const;
```

setEventData Stores event data that is specific to a particular notification. The existence and type of the event data is documented with the notification IDs in the definition of the derived notifier class.

```
INotificationEvent& setEventData( const IEventData& eventData);
```

setNotifierAttrChanged

Indicates that the notification event is a change in one of the notifier's attributes.

```
INotificationEvent&
   setNotifierAttrChanged( Boolean notifierAttrchanged = true);
```

setObserverData

Stores observer data in the notification event. The observer provides this data when it registers with a notifier by calling the INotifier::addObserver protected member function.

```
INotificationEvent& setObserverData( const IEventData& observerData);
```

Inherited Public Functions

IBase		
asDebugInfo	**messageFile**	**setMessageFile**
asString	**messageText**	**version**

Inherited Protected Data

IBase		
recoverable	**unrecoverable**	

INotifier

Derivation

IBase
  IVBase
    INotifier

Inherited By

IStandardNotifier
IWindow

Header File

inotify.hpp

Members

Member	Page	Member	Page
Constructor	408	notifyObservers	408
addObserver	410	observerList	410
disableNotification	408	removeAllObservers	410
enableNotification	408	removeObserver	410
isEnabledForNotification	408	~INotifier	408

The class INotifier defines the notification protocol that objects that support observation must supply. Because this class is an abstract base class, you cannot construct objects of this class. All IBM User Interface Class Library window classes inherit the notification process from INotifier.

You can implement a notification protocol in the following way:

- Derive a class from the IStandardNotifier class which inherits from INotifier for a direct implementation of the INotifier protocol
- Derive from the INotifier class and implement your own notification protocol

Because IWindow inherits from and implements the INotifier protocol, IWindow provides a visual implementation. IStandardNotifier inherits from INotifier and can be used for any generic notifier, visual or not. You might want to derive your classes from IStandardNotifier if you are providing a nonvisual notifier.

INotifier objects define INotificationIds for each notification that the derived class provides. You should document the details of these notifications, including any notifier data, within the description of the notification IDs of the derived class definition.

INotifier

INotifier objects notify their observers of all events after the observer requests notification by calling INotifier::addObserver. The observer object must check the notification ID and process the events it is interested in.

PM See *Designing Parts For Fun and Profit* for more information on part construction.

Public Functions

Constructor and Destructor

This class is an abstract base class therefore objects cannot be constructed.

Constructor `INotifier();`

Destructor `virtual ~INotifier();`

Notification Members

Use these members to affect the ability of INotifier to notify observers of events.

disableNotification

Stops the notifier from sending notifications to its observers.

```
virtual INotifier& disableNotification() = 0;
```

enableNotification

Starts the notifier sending notifications to its observers. This function can be overridden by derived classes to perform customized notification that your application might need. For instance, one of your function methods may require that a data base be accessible before processing a retrieve function.

```
virtual INotifier& enableNotification( Boolean enable = true) = 0;
```

isEnabledForNotification

Returns true if a notifier can send notifications to its observers.

```
virtual Boolean isEnabledForNotification() const = 0;
```

Observer Notification

These members notify observers of a change in a notifier.

notifyObservers

Notifies all observers in a notifier's list of observers. Each observer receives a notification event containing the identity of the notifier, the notification ID, and any optional data provided by the specific notifier object.

Note: A public and a protected version of notifyObservers are provided for convenience. The protected version does not require the caller to construct an INotificationEvent to call it. In this case, the construction of the INotificationEvent occurs in the code of the protected notifyObservers function.

```
virtual INotifier&
   notifyObservers( const INotificationEvent& event) = 0;
```

Inherited Public Functions

IVBase		
asDebugInfo	asString	

IBase		
asDebugInfo	**messageFile**	**setMessageFile**
asString	**messageText**	**version**

Protected Functions

Observer Addition and Removal

IObserver objects use these members to add and remove themselves from the notifier's collection.

addObserver Adds an observer to the notifier's list of observers.

```
virtual INotifier& addObserver( IObserver& observer,
    const IEventData& userData) = 0;
```

observerList Returns the list of observers. If the observer list does not exist, the derived notifier class must create it before calling this member function.

```
virtual IObserverList& observerList() const = 0;
```

removeAllObservers

Removes all observers from the notifier's list of observers.

```
virtual INotifier& removeAllObservers() = 0;
```

removeObserver

Removes an observer from the notifier's list of observers.

```
virtual INotifier& removeObserver( IObserver& observer) = 0;
```

Observer Notification

These members notify observers of a change in a notifier.

notifyObservers

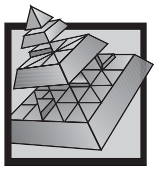
Notifies all observers in a notifier's list of observers. Each observer receives a notification event containing the identity of the notifier, the notification ID, and any optional data provided by the specific notifier object.

Note: A public and a protected version of notifyObservers are provided for convenience. The protected version does not require the caller to construct an INotificationEvent to call it. In this case, the construction of the INotificationEvent occurs in the code of the protected notifyObservers function.

```
virtual INotifier& notifyObservers( const INotificationId& id) = 0;
```

Inherited Protected Data

IBase		
recoverable	**unrecoverable**	



IObserver

Derivation

IBase
  IVBase
    IObserver

Inherited By None.

Header File iobservr.hpp

Members

Member	Page	Member	Page
Constructor	412	stopHandlingNotificationsFor	412
dispatchNotificationEvent	412	~IObserver	411
handleNotificationsFor	411		

The IObserver class is the abstract base class for all objects that are to be notified of changes in the state of other objects in the system. You can derive objects that require notification from this class and implement the function dispatchNotificationEvent to process specific events.

Public Functions

Constructor and Destructor

Only derived classes can create objects of this class. To enforce this, the only constructor has protected access.

Destructor
```
virtual ~IObserver();
```

Event Dispatching

Use these members to evaluate events and determine if it is appropriate for an observer object to process it. They also attach the observer to and detach the observer from the INotifier object.

handleNotificationsFor

Attaches the observer to the INotifier object argument. The observer is notified of events after the notifier object has been enabled for notifications.

```
virtual IObserver& handleNotificationsFor( INotifier& notifier,
    const IEventData& userData = IEventData ( ));
```

stopHandlingNotificationsFor

Detaches the observer from the argument INotifier object.

```
virtual IObserver&
   stopHandlingNotificationsFor( INotifier& notifier);
```

Inherited Public Functions

IVBase		
asDebugInfo	asString	

IBase		
asDebugInfo	**messageFile**	**setMessageFile**
asString	**messageText**	**version**

Protected Functions

Constructor

Only derived classes can create objects of this class. To enforce this, the only constructor has protected access.

Constructor The default constructor.

```
IObserver();
```

Event Dispatching

Use these members to evaluate events and determine if it is appropriate for an observer object to process it. They also attach the observer to and detach the observer from the INotifier object.

dispatchNotificationEvent

Notifies an observer of an event in a notification enabled object. The notification also includes event specific information.

```
virtual IObserver&
   dispatchNotificationEvent( const INotificationEvent& event) = 0;
```

Inherited Protected Data

IBase		
recoverable	**unrecoverable**	

IObserverList

Derivation

IBase
  IVBase
    IObserverList

Inherited By None.

Header File iobslist.hpp

Members

Member	Page	Member	Page
Constructor	413	numberOfElements	414
add	414	remove	414
elementAt	414	removeAll	414
isEmpty	414	removeAt	414
notifyObservers	414	~IObserverList	413

The IObserverList class provides the interface for a list of IObserver objects. This class implements the list of observers as an ordered list that can be traversed with cursor logic.

Public Functions

Constructor and Destructor

You can construct and destruct objects of this class.

Constructor

```
IObserverList();
```

You may only construct objects of this class using the default constructor that takes no arguments.

Destructor

```
virtual ~IObserverList();
```

Observer Addition and Removal

Use these members to add, remove, and find IObserver objects in the observer list's collection.

add
Adds an observer to the end of the list.

```
virtual Boolean add( IObserver& observer, void* userData);
```

elementAt
Returns an observer from the list using the specified cursor object.

```
virtual IObserver& elementAt( const Cursor& cursor) const;
```

isEmpty
Returns true if there are no observers in the list.

```
Boolean isEmpty() const;
```

numberOfElements
Returns the number of observers in the list.

```
unsigned long numberOfElements() const;
```

remove
Removes the specified observer from the list.

```
virtual IObserverList& remove( const IObserver& observer);
```

removeAll
Removes all observers from the list.

```
virtual IObserverList& removeAll();
```

removeAt
Removes an observer at the specified cursor location from the list.

```
virtual IObserverList& removeAt( const Cursor& cursor);
```

Observer Notification

These members notify observers of a change in a notifier.

notifyObservers
Traverses the list of observers and calls each member’s dispatchNotificationEvent function passing a specified notification event object.

```
IObserverList& notifyObservers( const INotificationEvent& event);
```

Inherited Public Functions

IVBase		
asDebugInfo	asString	

IBase		
asDebugInfo	**messageFile**	**setMessageFile**
asString	**messageText**	**version**

Inherited Protected Data

IBase		
recoverable	**unrecoverable**	

Nested Classes

IObserverList contains the following nested classes:

IObserverList::Cursor (see page 417)

IObserverList::Cursor

Derivation

IBase
  IVBase
    IObserverList::Cursor

Inherited By None.

Header File iobslist.hpp

Members

Member	Page	Member	Page
Constructor	417	setToLast	418
Cursor	417	setToNext	418
invalidate	417	setToPrevious	418
isValid	417	˜Cursor	417
setToFirst	418		

This is a nested cursor class used to iterate over the observers added to an INotifier.

Public Functions

Constructor and Destructor

You can construct and destruct objects of this class.

Constructor Create an IObserverList::Cursor by providing a reference to an IObserverlist.

```
Cursor( IObserverList& observerList);
```

Destructor

```
virtual ˜Cursor();
```

Cursor Movement

These members provide cursor movement operations.

invalidate Marks the cursor as invalid.

```
virtual void invalidate();
```

isValid Returns true if the cursor is on a valid observer.

```
virtual Boolean isValid() const;
```

setToFirst Set the cursor position to the first observer in the list.

```
virtual Boolean setToFirst();
```

setToLast Sets the cursor position to the last observer in the list.

```
virtual Boolean setToLast();
```

setToNext Advances the cursor position to the next observer in the list.

```
virtual Boolean setToNext();
```

setToPrevious
Sets the cursor position to the prior observer in the list.

```
virtual Boolean setToPrevious();
```

Inherited Public Functions

IVBase		
asDebugInfo	asString	

IBase		
asDebugInfo	**messageFile**	**setMessageFile**
asString	**messageText**	**version**

Inherited Protected Data

IBase		
recoverable	**unrecoverable**	

IOutOfMemory

Derivation

IException
  IResourceExhausted
    IOutOfMemory

Inherited By None.

Header File iexcbase.hpp

Members

Member	Page
Constructor	419
name	420

Objects of the IOutOfMemory class represent an exception. The User Interface Class Library's new_handler function creates an object of the IOutOfMemory class when heap memory is exhausted.

Public Functions

Constructor

You can construct objects of this class.

Constructor You can create objects of this class by doing the following:

- Using the constructor.

errorText	The text describing this particular error.
errorId	The identifier you want to associate with this particular error.
severity	Use the enumeration IException::Severity (p. 386) to specify the severity of the error. The default is unrecoverable.

- Using the macros discussed in IException (p. 379). The library provides these macros to make creating exceptions easier for you.

```
IOutOfMemory( const char* errorText, unsigned long errorId,
   Severity severity = IException::unrecoverable);
```

Exception Type

Use these members to determine the name (type) of the exception. This is used for logging out an exception object's error information.

name Returns the name of the object's class.

```
virtual const char* name() const;
```

Inherited Public Functions

IResourceExhausted		
name		

IException		
addLocation	locationAtIndex	setSeverity
appendText	locationCount	setText
assertParameter	logExceptionData	**setTraceFunction**
errorCodeGroup	name	terminate
errorId	setErrorCodeGroup	text
isRecoverable	setErrorId	textCount

Inherited Public Data

IException		
baseLibrary	**CLibrary**	**operatingSystem**

IOutOfSystemResource

Derivation

IException
  IResourceExhausted
    IOutOfSystemResource

Inherited By None.

Header File iexcbase.hpp

Members

Member	Page
Constructor	421
name	422

Objects of the IOutOfSystemResource class represent an exception. When a member function makes an operating system resource request that the system cannot satisfy, the member function creates and throws an object of the IOutOfSystemResource class.

Public Functions

Constructor

You can construct objects of this class.

Constructor You can create objects of this class by doing the following:

- Using the constructor.

errorText	The text describing this particular error.
errorId	The identifier you want to associate with this particular error.
severity	Use the enumeration IException::Severity (p. 386) to specify the severity of the error. The default is unrecoverable.

- Using the macros discussed in IException (p. 379). The library provides these macros to make creating exceptions easier for you.

```
IOutOfSystemResource( const char* errorText, unsigned long errorId,
   Severity severity = IException::unrecoverable);
```

Exception Type

Use these members to determine the name (type) of the exception. This is used for logging out an exception object's error information.

name Returns the name of the object's class.

```
virtual const char* name() const;
```

Inherited Public Functions

IResourceExhausted		
name		

IException		
addLocation	locationAtIndex	setSeverity
appendText	locationCount	setText
assertParameter	logExceptionData	**setTraceFunction**
errorCodeGroup	name	terminate
errorId	setErrorCodeGroup	text
isRecoverable	setErrorId	textCount

Inherited Public Data

IException		
baseLibrary	**CLibrary**	**operatingSystem**

IOutOfWindowResource

Derivation

IException
  IResourceExhausted
    IOutOfWindowResource

Inherited By None.

Header File iexcbase.hpp

Members

Member	Page
Constructor	423
name	424

Objects of the IOutOfWindowResource class represent an exception. When a member function makes a presentation (window) system resource request that the system cannot satisfy, the member function creates and throws an object of the IOutOfWindowResource class.

Public Functions

Constructor

You can construct objects of this class.

Constructor You can create objects of this class by doing the following:

- Using the constructor.

errorText	The text describing this particular error.
errorId	The identifier you want to associate with this particular error.
severity	Use the enumeration IException::Severity (p. 386) to specify the severity of the error. The default is unrecoverable.

- Using the macros discussed in IException (p. 379). The library provides these macros to make creating exceptions easier for you.

```
IOutOfWindowResource( const char* errorText, unsigned long errorId,
   Severity severity = IException::unrecoverable);
```

Exception Type

Use these members to determine the name (type) of the exception. This is used for logging out an exception object's error information.

name Returns the name of the object’s class.

```
virtual const char* name() const;
```

Inherited Public Functions

IResourceExhausted		
name		

IException		
addLocation	locationAtIndex	setSeverity
appendText	locationCount	setText
assertParameter	logExceptionData	**setTraceFunction**
errorCodeGroup	name	terminate
errorId	setErrorCodeGroup	text
isRecoverable	setErrorId	textCount

Inherited Public Data

IException		
baseLibrary	**CLibrary**	**operatingSystem**

IPair

Derivation

IBase
  IPair

Inherited By

IPoint
IRange
ISize

Header File

ipoint.hpp

Members

Member	Page	Member	Page
Constructor	426	operator -	427
asDebugInfo	426	operator -=	428
asString	426	operator /=	428
coord1	427	operator <	426
coord2	427	operator <=	426
distanceFrom	429	operator ==	426
dotProduct	429	operator >	426
maximum	428	operator >=	426
minimum	428	scaleBy	428
operator !=	426	scaledBy	428
operator %=	427	setCoord1	427
operator *=	427	setCoord2	427
operator +=	428	transpose	429

Objects of the IPair class are generic ordered pairs of coordinates. The class serves as the base for the following specific ordered pair classes:

- IPoint (p. 431)
- ISize (p. 461)
- IRange (p. 439)

This class provides basic utilities to get and set the two coordinate values. In addition, it provides a full set of comparison and mathematical operators to manipulate ordered pairs.

Public Functions

Comparison Operators

Use these members to compare one IPair object to another.

operator != True if either coordinate differs.

```
Boolean operator !=( const IPair& aPair) const;
```

operator < True if both coordinates are less than those of the specified *aPair*.

```
Boolean operator <( const IPair& aPair) const;
```

operator <= True if both coordinates are less than or equal.

```
Boolean operator <=( const IPair& aPair) const;
```

operator == True if both coordinates match those of the specified *aPair*.

```
Boolean operator ==( const IPair& aPair) const;
```

operator > True if both coordinates are greater than those of the specified *aPair*.

```
Boolean operator >( const IPair& aPair) const;
```

operator >= True if both coordinates are greater than or equal.

```
Boolean operator >=( const IPair& aPair) const;
```

Constructors

You can construct, copy, and assign objects of this class. This class uses the compiler-generated copy constructor and assignment operator to copy and assign IPair objects.

Constructors

```
IPair( Coord init);
IPair();
IPair( Coord coord1, Coord coord2);
```

Conversions

Use these members to return an IPair object in a different form.

asDebugInfo Converts the ordered pair to an IString (p. 469) containing a diagnostic representation of the object.

```
IString asDebugInfo() const;
```

asString Converts the ordered pair (a, b) to an IString((p. 469) "(a, b)").

```
IString asString() const;
```

operator - Returns an ordered pair whose coordinates are the difference between the corresponding coordinates of *pair1* and *pair2*.

When you use the unary format, it returns an ordered pair with negated coordinates.

```
IPair operator -() const;
```

Coordinates

Use these members to query and change the ordered pair of integers in an IPair object.

coord1 Obtains the value of the first coordinate.

```
Coord coord1() const;
```

coord2 Obtains the value of the second coordinate.

```
Coord coord2() const;
```

setCoord1 Sets the value of the first coordinate.

```
IPair& setCoord1( Coord coord1);
```

setCoord2 Sets the value of the second coordinate.

```
IPair& setCoord2( Coord coord2);
```

Manipulation

Use these members to alter the coordinate values of an IPair object. This includes both member and non-member arithmetic operators and members to scale the value of an IPair object.

operator %= Replaces the coordinates with the remainder when divided by those of the following specified parameter:

aPair
The library performs the remainder function between the corresponding coordinates, coord1 with coord1 of *aPair* and coord2 with coord2.

divisor
The library performs the remainder function between each coordinate and the *divisor*.

```
IPair& operator %=( long divisor);
IPair& operator %=( const IPair& aPair);
```

operator *= Multiplies the coordinates by those of the specified parameter:

aPair
The library performs the product function between the corresponding coordinates, coord1 with coord1 of *aPair* and coord2 with coord2.

multiplier
The library perform the product function between each coordinate and the *multiplier*.

```
IPair& operator *=( double multiplier);
IPair& operator *=( const IPair& aPair);
```

operator += Adds the coordinates of the specified *aPair* to the coordinates of an ordered pair.

```
IPair& operator +=( const IPair& aPair);
```

operator -= Subtracts the coordinates specified in *aPair* from the IPair coordinates.

```
IPair& operator -=( const IPair& aPair);
```

operator /= Divides the coordinates by those of the second specified parameter:

aPair
The library performs the quotient function between the corresponding coordinates, coord1 with coord1 of *aPair* and coord2 with coord2.

divisor
The library performs the product function between each coordinate and the *divisor*.

```
IPair& operator /=( const IPair& aPair);
IPair& operator /=( double divisor);
```

scaleBy Scales the X-coordinate by *xFactor*, the Y-coordinate by *yFactor*.

```
IPair& scaleBy( double xFactor, double yFactor);
```

scaledBy Same as IPair::scaleBy (p. 428), but returns a new IPair, leaving the original unmodified.

```
IPair scaledBy( double xFactor, double yFactor) const;
```

Minimum and Maximum

Use these members to determine the smaller or larger of two IPair objects.

maximum Returns an ordered pair whose coordinates are the maximum of the corresponding coordinates of the IPair and the specified IPair.

```
IPair maximum( const IPair& aPair) const;
```

minimum Returns an ordered pair whose coordinates are the minimum of the corresponding coordinates of the IPair and the specified IPair.

```
IPair minimum( const IPair& aPair) const;
```

Miscellaneous

These members are additional, unrelated members of the IPair class.

distanceFrom

Returns the distance from some other ordered pair.

```
double distanceFrom( const IPair& aPair) const;
```

dotProduct

Returns the dot product with another ordered pair.

```
long dotProduct( const IPair& aPair) const;
```

transpose

Swaps the coordinates of the ordered pair. The friend version of this function returns a new pair with transposed coordinates.

```
IPair& transpose();
```

Inherited Public Functions

IBase		
asDebugInfo	**messageFile**	**setMessageFile**
asString	**messageText**	**version**

Inherited Protected Data

IBase		
recoverable	**unrecoverable**	

Nested Type Definitions

Coord

```
typedef long Coord;
```

Type of the coordinate values (long integer).

IPoint

Derivation

IBase
  IPair
    IPoint

Inherited By None.

Header File ipoint.hpp

Members

Member	Page	Member	Page
Constructor	431	setY	432
asPOINTL	432	x	432
setX	432	y	432

Objects of the IPoint class represent points in two-dimensional space. In addition to all the functions inherited from its base class, IPair (p. 425), the IPoint class provides additional functions.

PM You can also construct objects of this class from a Presentation Manager Toolkit POINTL structure.

Public Functions

Constructors

You can create, copy, and assign objects of this class. This class uses the compiler-generated copy constructor and assignment operator to copy and assign IPoint objects.

Constructors

```
1   IPoint( const IPair& pair);
2   IPoint();
3   IPoint( Coord x, Coord y);
4   IPoint( const struct _POINTL& ptl);
```

Supported On:
PM

Conversions

Use these members to return an IPoint object in a different form.

asPOINTL Renders the point as a Presentation Manager Toolkit POINTL structure.

```
struct _POINTL asPOINTL() const;
```

Supported On:
PM

Coordinates

Use these members to query and change the x and y coordinates of an IPoint object.

setX Sets the point's X-coordinate.

```
IPoint& setX( Coord X);
```

setY Sets the point's Y-coordinate.

```
IPoint& setY( Coord Y);
```

x Returns the point's X-coordinate.

```
Coord x() const;
```

y Returns the point's Y-coordinate.

```
Coord y() const;
```

Inherited Public Functions

IPair		
asDebugInfo	operator !=	operator <=
asString	operator %=	operator ==
coord1	operator *=	operator >
coord2	operator +=	operator >=
distanceFrom	operator -	scaleBy
dotProduct	operator -=	scaledBy
maximum	operator /=	setCoord1
minimum	operator <	setCoord2

IBase		
asDebugInfo	**messageFile**	**setMessageFile**
asString	**messageText**	**version**

Inherited Protected Data

IBase		
recoverable	**unrecoverable**	

IPointArray

Derivation

IBase
  IPointArray

Inherited By None.

Header File iptarray.hpp

Members

Member	Page	Member	Page
Constructor	435	remove	436
add	436	resize	436
insert	436	reverse	436
operator !=	435	reversed	436
operator =	436	size	437
operator ==	435	~IPointArray	436
operator []	436		

The IPointArray class is used to represent an array of IPoint objects.

Public Functions

Comparisons

Use these members to compare two point arrays.

operator != Returns true if the arrays are not the same length or the points are not identical or both.

```
operator !=( const IPointArray& pointArray) const;
```

operator == Returns true if the arrays are the same length and have identical points.

```
Boolean operator ==( const IPointArray& pointArray) const;
```

Constructors and Destructor

You can construct, copy, and assign objects of this class.

Constructors

1
```
IPointArray( unsigned long dimension = 0, const IPoint* array = 0);
```

Use this function to construct a IPointArray object from two optional arguments. The first argument specifies the length of the array and the second argument is a pointer to an array of IPoint objects. The array of IPoints are used to initialize the IPointArray object. If a pointer to an array of IPoint objects is specified, it is assumed that the IPoint array has at least as many elements as the array dimension specified.

2
```
IPointArray( const IPointArray& pointArray);
```

Use this function to construct a IPointArray object from an existing IPointArray object.

operator =

```
IPointArray& operator =( const IPointArray& pointArray);
```

Use this function to assign one IPointArray object to another. The target IPointArray object is grown or shrunk to the size of the source IPointArray object.

Destructor

```
~IPointArray();
```

Data Access

Use these members to access attributes of objects of this class.

add

Adds a point to the end of the array.

```
IPointArray& add( const IPoint& point);
```

insert

Inserts a point before the index specified.

```
IPointArray& insert( unsigned long index, const IPoint& point);
```

operator []

Returns a reference to the point at the specified index.

```
IPoint& operator []( unsigned long index);
const IPoint& operator []( unsigned long index) const;
```

remove

Removes a point at the specified index.

```
IPointArray& remove( unsigned long index);
```

resize

Increases or decreases the size of the array. New points are initialized to 0,0.

```
IPointArray& resize( unsigned long newsize);
```

reverse

Reverses the elements in the array.

```
IPointArray& reverse();
```

reversed

Returns a copy of the point array with its elements reversed.

```
IPointArray reversed() const;
```

size
Returns the dimension of the array.

```
unsigned long size() const;
```

Inherited Public Functions

IBase		
asDebugInfo	**messageFile**	**setMessageFile**
asString	**messageText**	**version**

Inherited Protected Data

IBase		
recoverable	**unrecoverable**	

IRange

Derivation

```
IBase
  IPair
    IRange
```

Inherited By None.

Header File ipoint.hpp

Members

Member	Page	Member	Page
Constructor	439	setLowerBound	439
includes	440	setUpperBound	440
lowerBound	439	upperBound	440

Objects of the IRange class represent a range of IPair::Coord values between a specified lower and upper bound (inclusive).

Public Functions

Constructors

You can construct, copy, and assign objects of this class. This class uses the compiler-generated copy constructor and assignment operator to copy and assign IRange objects.

Constructors

```
IRange( Coord lower, Coord upper);
IRange();
IRange( const IPair& aPair);
```

Coordinates

Use these members to query and change the ordered pair of integers in an IRange object.

lowerBound Returns the lower bound of the range.

```
Coord lowerBound() const;
```

setLowerBound

Sets the lower bound of the range.

```
IRange& setLowerBound( Coord lower);
```

setUpperBound

Sets the upper bound of the range.

```
IRange& setUpperBound( Coord upper);
```

upperBound Returns the upper bound of the range.

```
Coord upperBound() const;
```

Testing

Use these members to test coordinate values.

includes Returns true if the range contains the specified coordinate value.

```
Boolean includes( Coord aValue) const;
```

Inherited Public Functions

IPair		
asDebugInfo	operator !=	operator <=
asString	operator %=	operator ==
coord1	operator *=	operator >
coord2	operator +=	operator >=
distanceFrom	operator -	scaleBy
dotProduct	operator -=	scaledBy
maximum	operator /=	setCoord1
minimum	operator <	setCoord2

IBase		
asDebugInfo	**messageFile**	**setMessageFile**
asString	**messageText**	**version**

Inherited Protected Data

IBase		
recoverable	**unrecoverable**	

IRectangle

Derivation

IBase
  IRectangle

Inherited By

None.

Header File

irect.hpp

Members

Member	Page	Member	Page
Constructor	443	minXMinY	449
area	444	minY	445
asDebugInfo	444	moveBy	445
asRECTL	444	movedBy	445
asString	444	movedTo	446
bottom	449	moveTo	446
bottomCenter	449	operator !=	442
bottomLeft	449	operator &	447
bottomRight	449	operator &=	447
center	450	operator ==	442
centerAt	445	operator |	447
centeredAt	445	operator |=	448
centerXCenterY	448	right	450
centerXMaxY	448	rightCenter	450
centerXMinY	448	scaleBy	446
contains	451	scaledBy	446
expandBy	445	shrinkBy	446
expandedBy	445	shrunkBy	446
height	444	size	445
intersects	451	sizeBy	447
left	450	sizedBy	447
leftCenter	450	sizedTo	447
maxX	444	sizeTo	447
maxXCenterY	448	top	450
maxXMaxY	448	topCenter	450
maxXMinY	448	topLeft	450
maxY	444	topRight	450
minX	445	validate	451
minXCenterY	448	width	445
minXMaxY	448		

IRectangle

Objects of the IRectangle class represent a rectangular area defined by two points that form opposite corners of the rectangle. These two points are referred to as the minimum and maximum points.

IRectangle objects are designed to be usable independently of the coordinate system in use. The minimum, or origin, is defined as the point with the lowest coordinate values. Therefore, in a coordinate space where 0,0 is the upper left and increasing a point's coordinate value moves it to the right and down, the minimum point of an IRectangle will be the top left corner and the maximum corner the lower right corner. Conversely, in a coordinate space where 0,0 is the lower left corner and increasing a point's coordinate value moves it to the right and up, the minimum corner of an IRectangle will be the lower left, and the maximum corner the top right.

IRectangle provides some member functions which use the terms "right", "left", "top", and "bottom". These are synonyms for the functions defined in terms of minimum and maximum corners. The directional orientation for the right/left/top/bottom functions is correct for a coordinate space where 0,0 is the lower left corner. For other coordinate systems, left and bottom are the sides of the rectangle with the lowest coordinate value, and top and right the sides with the highest.

Mathematically, a rectangle includes all the points on the lines that intersect its minimum corner. It does not include the points that lie on its edges that do not intersect the origin. This is important when you are doing detailed graphics work. For example, a rectangle specified as having a minimum of 0,0 and maximum of 10,20 will include the points 0,0 though 0,19 but will not include 0,20. Similarly, the points 1,0 through 9,0 are contained but 10,0 is not.

Various graphics and windowing classes, as well as their member functions, use rectangles.

PM You can also construct objects of this class by providing a Presentation Manager Toolkit RECTL structure.

Public Functions

Comparisons

Use these members to compare two rectangles for equality or inequality.

operator != If the rectangles differ, true is returned.

```
Boolean operator !=( const IRectangle& rectangle) const;
```

operator == If the two rectangles are identical, true is returned. Identity of rectangles means that the two defining points are the same.

```
Boolean operator ==( const IRectangle& rectangle) const;
```

Constructors

You can construct, copy, and assign objects of this class.

Note: The library constructs rectangles by taking the two points that are given (or implied) as opposite corners. The minimum point, or origin, is set to be the lesser of the two points. This ensures that internally the origin and corner points always are such that the origin is less than or equal to the corner.

Constructors

1
```
IRectangle( const IPair& pair);
```

Constructs a rectangle with corners at 0,0 and the specified location. The lower of **aPair** and 0,0 will be the origin of the rectangle.

2
```
IRectangle();
```

Constructs a rectangle at (0,0),(0,0).

3
```
IRectangle( const IPoint& point1, const IPoint& point2);
```

Constructs a rectangle from two points at opposite corners.

4
```
IRectangle( const IPoint& point, const ISize& size);
```

Creates a rectangle from a point and size. The maximum point is calculated by adding the width of the size to the x coordinate of the given point and adding the height of the size to the y coordinate of the given point.

5
```
IRectangle( Coord point1X, Coord point1Y,
    Coord point2X, Coord point2Y);
```

Constructs a rectangle from four values representing coordinates of the corners.

point1X
The X coordinate of point 1.

point1Y
The Y coordinate of point 1.

point2X
The X coordinate of point 2.

point2Y
The Y coordinate of point 2.

6
```
IRectangle( const struct _RECTL& rectl);
```

Supported On:
PM

Constructs a rectangle from a PM toolkit RECTL structure.

7 `IRectangle( Coord width, Coord height);`

Constructs a rectangle of the specified size. The size is specified as

width
The width of the rectangle.

height
The height of the rectangle.

Conversions

Use these members to convert a rectangle into various formats.

asDebugInfo Renders the rectangle as a diagnostic representation.

```
IString asDebugInfo() const;
```

asRECTL Converts the rectangle into a system dependent structure.

```
struct _RECTL asRECTL() const;
```

Supported On:
PM

PM Renders the rectangle as a Presentation Manager Toolkit RECTL structure.

asString Renders the rectangle as an IString("IRectangle(x1,y1,x2,y2)").

```
IString asString() const;
```

Dimensions

Use these members to obtain information about a rectangle's size.

area Returns the area of the rectangle. The area is determined by multiplying the width of the rectangle by the height. For example, a rectangle defined from IPoint(1,1), IPoint(10,20) would have an area of 9*19, or 171.

```
Coord area() const;
```

height Returns the height of the rectangle. The height is determined by subtracting the y coordinate of the minimum point from the y coordinate of the maximum point.

```
Coord height() const;
```

maxX Returns the X-coordinate of the vertical line that is opposite the origin of the rectangle.

```
Coord maxX() const;
```

maxY Returns the Y-coordinate of the horizontal line opposite the origin.

```
Coord maxY() const;
```

minX Returns the X-coordinate of the vertical line that passes though the origin.

```
Coord minX() const;
```

minY Returns the Y-coordinate of the horizontal line that passes though the origin of the rectangle.

```
Coord minY() const;
```

size Returns the ISize(width, height).

```
ISize size() const;
```

width Returns the width of the rectangle. The width is determined by subtracting the x coordinate of the minimum point from the x coordinate of the maximum point.

```
Coord width() const;
```

Manipulation

Use these members to modify the rectangle, changing its size, proportions, or location.

centerAt Moves the rectangle so that its center is at the specified point.

```
IRectangle& centerAt( const IPoint& point);
```

centeredAt Same as IRectangle::centerAt (p. 445), but returns a new rectangle, leaving the original unmodified.

```
IRectangle centeredAt( const IPoint& point) const;
```

expandBy Moves the corners of the rectangle outward from the center by the specified amount. The specified amount can be either a scalar (long integer) or a point.

```
IRectangle& expandBy( const IPair& pair);
IRectangle& expandBy( Coord coord);
```

expandedBy Same as IRectangle::expandBy (p. 445), but returns a new rectangle, leaving the original unmodified.

```
IRectangle expandedBy( const IPair& pair) const;
IRectangle expandedBy( Coord coord) const;
```

moveBy Moves the rectangle by the amount specified by *aPair*.

```
IRectangle& moveBy( const IPair& pair);
```

movedBy Same as IRectangle::moveBy (p. 445), but returns a new rectangle, leaving the original unmodified.

```
IRectangle movedBy( const IPair& pair) const;
```

movedTo Same as IRectangle::moveTo (p. 446), but returns a new rectangle, leaving the original unmodified.

```
IRectangle movedTo( const IPoint& point) const;
```

moveTo Moves the rectangle so that its origin corner is at the specified point.

```
IRectangle& moveTo( const IPoint& point);
```

scaleBy Scales the rectangle by the specified amount. Scaling a rectangle multiplies its coordinates by the scale amount.

1
```
IRectangle& scaleBy( Coord coord);
```

Scales by a long integer value.

2
```
IRectangle& scaleBy( const IPair& pair);
```

Scales by a point specifying the amounts in the X- and Y-axis directions.

3
```
IRectangle& scaleBy( double factor);
```

Scales by a double value.

4
```
IRectangle& scaleBy( double xfactor, double yfactor);
```

Scales by a pair of doubles. The function uses the first double to scale in the X-axis direction, the second in the Y-axis direction.

scaledBy Same as IRectangle::scaleBy (p. 446), but returns a new rectangle, leaving the original unmodified.

```
IRectangle scaledBy( double xfactor, double yfactor) const;
IRectangle scaledBy( const IPair& pair) const;
IRectangle scaledBy( Coord coord) const;
IRectangle scaledBy( double factor) const;
```

shrinkBy Moves the corners of the rectangle inward toward the center by the specified amount, either a scalar or a point.

Note: shrinkBy(anAmount) is always equivalent to expandBy(- anAmount), and vice versa.

```
IRectangle& shrinkBy( Coord coord);
IRectangle& shrinkBy( const IPair& pair);
```

shrunkBy Same as IRectangle::shrinkBy (p. 446), but returns a new rectangle, leaving the original unmodified.

```
IRectangle shrunkBy( const IPair& pair) const;
IRectangle shrunkBy( Coord coord) const;
```

sizeBy Scales the rectangle by the specified value, leaving the rectangle at the same location because the bottom-left point remains fixed.

1
```
IRectangle& sizeBy( const IPair& pair);
```

Scales by a pair of integer scalars specifying different factors in the X-axis and Y-axis directions.

2
```
IRectangle& sizeBy( Coord factor);
```

Scales by the same integer factor in both the X-axis and Y-axis directions.

3
```
IRectangle& sizeBy( double factor);
```

Scales by the same double factor in both the X-axis and Y-axis directions.

4
```
IRectangle& sizeBy( double xfactor, double yfactor);
```

Scales by two doubles specifying factors in the X-axis and Y-axis directions, respectively.

sizedBy Same as IRectangle::sizeBy (p. 447), but returns a new rectangle, leaving the original unmodified.

```
IRectangle sizedBy( double factor) const;
IRectangle sizedBy( const IPair& pair) const;
IRectangle sizedBy( Coord factor) const;
IRectangle sizedBy( double xfactor, double yfactor) const;
```

sizedTo Same as IRectangle::sizeTo (p. 447), but returns a new rectangle, leaving the original unmodified.

```
IRectangle sizedTo( const IPair& pair) const;
```

sizeTo Sizes the rectangle to the specified size.

```
IRectangle& sizeTo( const IPair& pair);
```

Manipulation Operators

Use these members to find a rectangle's union and intersection with another rectangle.

operator & Returns a rectangle representing the intersection of the specified rectangles.

```
IRectangle operator &( const IRectangle& rectangle) const;
```

operator &= Resets the rectangle to its intersection with the specified rectangle.

```
IRectangle& operator &=( const IRectangle& rectangle);
```

operator | Returns the rectangle representing the union of the specified rectangles. This is the smallest rectangle that encompasses both specified rectangles.

```
IRectangle operator |( const IRectangle& rectangle) const;
```

operator |= Resets the rectangle to its union with the specified rectangle.

```
IRectangle& operator |=( const IRectangle& rectangle);
```

Points

Use these members to access points on or within the rectangle. You can query any of nine points on a rectangle's perimeter or its center by using these members.

centerXCenterY

Returns the X- and Y-coordinates of the center point of the rectangle.

```
IPoint centerXCenterY() const;
```

centerXMaxY Returns the X- and Y-coordinates of the center point of the horizontal line opposite the origin of the rectangle.

```
IPoint centerXMaxY() const;
```

centerXMinY Returns the X- and Y-coordinates of the center point of the horizontal line passing through the origin of the rectangle.

```
IPoint centerXMinY() const;
```

maxXCenterY

Returns the X- and Y-coordinates of the center point of the vertical line opposite the origin of the rectangle.

```
IPoint maxXCenterY() const;
```

maxXMaxY Returns the X- and Y-coordinates of the corner of the rectangle opposite the origin.

```
IPoint maxXMaxY() const;
```

maxXMinY Returns the X- and Y-coordinates of the corner of the rectangle at the other end of the horizontal line passing though the origin.

```
IPoint maxXMinY() const;
```

minXCenterY Returns the X- and Y-coordinates of the center of the vertical line that passes through the origin.

```
IPoint minXCenterY() const;
```

minXMaxY Returns the X- and Y-coordinates of the corner of the rectangle at the other end of the vertical line passing though the origin.

```
IPoint minXMaxY() const;
```

minXMinY Returns the X- and Y-coordinates of the origin corner of the rectangle.

```
IPoint minXMinY() const;
```

Synonyms

Use these members when you are working in a coordinate space where 0,0 is the lower left corner. In this case the right/left/top/bottom orientation is correct. You can still use these members in other coordinate systems, but left and bottom are the sides of the rectangle with the lowest coordinate value, and top and right are the sides with the highest.

The following table lists the members and synonyms defined for them:

Function	**Synonym**
minXMinY	bottomLeft
minXCenterY	leftCenter
minXMaxY	topLeft
centerXMinY	bottomCenter
centerXCenterY	center
centerXMaxY	topCenter
maxXMinY	bottomRight
maxXCenterY	rightCenter
maxXMaxY	topRight
minX	left
minY	bottom
maxX	right
maxY	top

bottom Returns the Y-coordinate of the horizontal line that forms the bottom of the rectangle. This is an alias for IRectangle::minY (p. 445).

```
Coord bottom() const;
```

bottomCenter

Returns the X- and Y-coordinates of the bottom-center point of the rectangle. This is an alias for IRectangle::centerXMinY (p. 448).

```
IPoint bottomCenter() const;
```

bottomLeft Returns the X- and Y-coordinates of the bottom-left corner of the rectangle This is an alias for IRectangle::minXMinY (p. 449).

```
IPoint bottomLeft() const;
```

bottomRight Returns the X- and Y-coordinates of the bottom-right corner of the rectangle. This is an alias for IRectangle::maxXMinY (p. 448).

```
IPoint bottomRight() const;
```

center Returns the X- and Y-coordinates of the center point of the rectangle. This is an alias for IRectangle::centerXCenterY (p. 448).

```
IPoint center() const;
```

left Returns the X-coordinate of the vertical line that forms the left side of the rectangle. This is an alias for IRectangle::minX (p. 445).

```
Coord left() const;
```

leftCenter Returns the X- and Y-coordinates of the left-center point of the rectangle. This is an alias for IRectangle::minXCenterY (p. 448).

```
IPoint leftCenter() const;
```

right Returns the X-coordinate of the vertical line that forms the right side of the rectangle. This is an alias for IRectangle::maxX (p. 444).

```
Coord right() const;
```

rightCenter Returns the X- and Y-coordinates of the right-center point of the rectangle. This is an alias for IRectangle::maxXCenterY (p. 448).

```
IPoint rightCenter() const;
```

top Returns the Y-coordinate of the horizontal line that forms the top of the rectangle. This is an alias for IRectangle::maxY (p. 444).

```
Coord top() const;
```

topCenter Returns the X- and Y-coordinates of the top-center point of the rectangle. This is an alias for IRectangle::centerXMaxY (p. 448).

```
IPoint topCenter() const;
```

topLeft Returns the X- and Y-coordinates of the top-left corner of the rectangle. This is an alias for IRectangle::minXMaxY (p. 448).

```
IPoint topLeft() const;
```

topRight Returns the X- and Y-coordinates of the top-right corner of the rectangle. This is an alias for IRectangle::maxXMaxY (p. 448).

```
IPoint topRight() const;
```

Testing

Use these members to test various attributes of a rectangle.

contains If the rectangle contains the specified point or rectangle, true is returned. A point is contained by a rectangle if its coordinates are greater than or equal to the minimum point of the rectangle and less than the maximum point. A rectangle is contained within another rectangle if its minimum point is greater than or equal to the containing rectangle's minimum point and its maximum point is less than or equal to the containing rectangle's maximum point.

```
Boolean contains( const IPoint& point) const;
Boolean contains( const IRectangle& rectangle) const;
```

intersects If the rectangle and specified rectangle overlap, true is returned.

```
Boolean intersects( const IRectangle& rectangle) const;
```

Inherited Public Functions

IBase		
asDebugInfo	**messageFile**	**setMessageFile**
asString	**messageText**	**version**

Protected Functions

Implementation

These members are used internally to implement the class.

validate Corrects an invalid rectangle after creation, expansion, or intersection.

```
IRectangle& validate();
```

Inherited Protected Data

IBase		
recoverable	**unrecoverable**	

Nested Type Definitions

Coord

```
typedef IPair::Coord Coord;
```

Type of the coordinate values; this must match the type of the coordinates supported by the IPair class.

IRefCounted

Derivation

IBase
  IVBase
    IRefCounted

Inherited By

IDMItem	IThreadFn
IDMOperation	ITimerFn
IStringGeneratorFn	

Header File irefcnt.hpp

Members

Member	Page	Member	Page
Constructor	454	useCount	454
addRef	453	~IRefCounted	454
removeRef	454		

The IRefCounted class is a public base class for any class that is reference counted. Such inheritance conveys the functional characteristics of maintaining a count of all references to the object and deferring destruction until all such references are destroyed.

By necessity, you can only allocate objects of this class in free store. The library enforces this by making the destructor for this class protected. As a result, the library only allows IRefCounted::removeRef (p. 454) and derived class destructors to call IRefCounted::IRefCounted. Derived classes should make their destructors protected also.

Typically, you use this class in conjunction with the corresponding IReference<T> (p. 455), where T is a derived class of IRefCounted.

Public Functions

Reference Counting

Use these members to manage the object's reference count.

addRef Adds a reference to the referred-to object.

```
virtual void addRef();
```

removeRef Removes a reference to the referred-to object. When the reference count goes to 0, this function deletes the referred-to object.

```
virtual void removeRef();
```

useCount Returns the use count for the referred-to object.

```
unsigned useCount() const;
```

Inherited Public Functions

IVBase		
asDebugInfo	asString	

IBase		
asDebugInfo	**messageFile**	**setMessageFile**
asString	**messageText**	**version**

Protected Functions

Constructor and Destructor

These members are protected.

Constructor `IRefCounted();`

Destructor `~IRefCounted();`

Inherited Protected Data

IBase		
recoverable	**unrecoverable**	

IReference

Derivation

IBase
  IReference

Inherited By None.

Header File irefcnt.hpp

Members

Member	Page	Member	Page
Constructor	456	operator =	456
operator *	457	operator T *	457
operator ->	457	˜IReference	456

The template class IReference is derived from classes that serve as references. Objects of such classes serve as smart pointers to objects of the referenced class. Creating objects of this class increments the use count of the referenced object. Destruction of the object causes the use count of the referenced object to be decremented.

Typically, this class is referenced explicitly only as a public base class of the class that provides the additional capability of the reference class. For example:

```
class Foo { .. };
class FooRef : public IReference<Foo> {
// Additional FooRef functions...
};
```

The reference-counted class provided as the template argument is derived from the class IRefCounted (p. 453). It must have the member functions IRefCounted::addRef (p. 453) and IRefCounted::removeRef (p. 454) with equivalent semantics.

To construct an IReference, you must provide a pointer to an object of the referenced (reference-counted) class. All constructors of the real reference class (derived from IReference<T>) must provide such a pointer. Otherwise, the reference class has no additional responsibilities.

Notes:

1. The semantics of such reference or referent classes can have subtle complexities. The reference or the referent might behave in an extraordinary fashion.
2. A class can also serve as a reference by having as a data member an IReference<T> object.
3. All members of the IReference class are public in order to permit the usage described in item 2.

Customization (Template Argument)

IReference is a template class that is instantiated with the following template argument:

T	Specifies the name of the class of objects to which template class objects refer.

Public Functions

Constructors and Destructor

You can construct, destruct, copy, and assign objects of this class.

Constructors

1
```
IReference( T* p = 0);
```

You can construct objects of this class by using this primary constructor which accepts a pointer to an instance of the referenced class. This also serves as the default constructor (defaulting the pointer parameter to 0).

2
```
IReference( const IReference < T >& source);
```

You can construct objects of this class by using this copy constructor which the library provides to ensure that the reference counts for both the source and target referents are maintained properly.

operator = The assignment operator. You can assign one IReference to another or you can assign a pointer to the referenced type.

```
IReference < T >& operator =( const IReference < T >& source);
IReference < T >& operator =( T* p);
```

Destructor The destructor ensures that the referenced object is de-referenced.

```
~IReference();
```

Operators

Use these members to access the referenced object. Their effect is to make an IReference usable, similar to a normal pointer.

operator * Pointer de-reference operator that provides access to the referenced object.

```
T& operator *() const;
```

operator -> Pointer operator that provides access to the referenced object.

```
T* operator ->() const;
```

operator T * Returns the referent.

```
operator T *() const;
```

Inherited Public Functions

IBase		
asDebugInfo	**messageFile**	**setMessageFile**
asString	**messageText**	**version**

Inherited Protected Data

IBase		
recoverable	**unrecoverable**	

IResourceExhausted

Derivation

IException
  IResourceExhausted

Inherited By

IOutOfMemory
IOutOfSystemResource
IOutOfWindowResource

Header File iexcbase.hpp

Members

Member	Page
Constructor	459
name	460

Objects of the IResourceExhausted class represent an exception. When a member function makes a resource request of the operating system or the presentation system that it cannot satisfy, the member function creates and throws an object of the IResourceExhausted class or one of its derived classes. IResourceExhausted is the generic out-of-resource class. Member functions use IResourceExhausted whenever its derived classes, which are for specific out-of-resource cases, are not applicable.

The derived classes for IResourceExhausted are:

IOutOfMemory (p. 419)
IOutOfSystemResource (p. 421)
IOutOfWindowResource (p. 423)

Public Functions

Constructor

You can construct objects of this class.

Constructor You can create objects of this class by doing the following:

- Using the constructor.

  errorText The text describing this particular error.

errorId	The identifier you want to associate with this particular error.
severity	Use the enumeration IException::Severity (p. 386) to specify the severity of the error. The default is unrecoverable.

- Using the macros discussed in IException (p. 379). The library provides these macros to make creating exceptions easier for you.

```
IResourceExhausted( const char* errorText, unsigned long errorId,
   Severity severity = IException::unrecoverable);
```

Exception Type

Use these members to determine the name (type) of the exception. This is used for logging out an exception object's error information.

name Returns the name of the object’s class.

```
virtual const char* name() const;
```

Inherited Public Functions

IException		
addLocation	locationAtIndex	setSeverity
appendText	locationCount	setText
assertParameter	logExceptionData	**setTraceFunction**
errorCodeGroup	name	terminate
errorId	setErrorCodeGroup	text
isRecoverable	setErrorId	textCount

Inherited Public Data

IException		
baseLibrary	**CLibrary**	**operatingSystem**

ISize

Derivation

IBase
  IPair
    ISize

Inherited By None.

Header File ipoint.hpp

Members

Member	Page	Member	Page
Constructor	461	setHeight	462
asSIZEL	461	setWidth	462
height	462	width	462

Objects of the ISize class use their coordinates to represent a rectangular size, in horizontal and vertical dimensions.

PM You can also construct objects of this class using:

- A Presentation Manager Toolkit SIZEL structure.
- A Presentation Manager Toolkit RECTL structure; in this case, the resulting ISize object represents the size of the RECTL.

Public Functions

Constructor

You can construct, copy, and assign objects of this class. This class uses the compiler-generated copy constructor and assignment operator to copy and assign ISize objects.

Constructors

```
ISize( const IPair& pair);
ISize();
ISize( Coord width, Coord height);
ISize( const SIZEL& sizl);
ISize( const struct _RECTL& rcl);
```

Conversions

Use these members to return an ISize object in a different form.

asSIZEL Returns the ISize as a Presentation Manager Toolkit SIZEL structure.

```
SIZEL asSIZEL() const;
```

Coordinates

Use these members to query and change the ordered pair of integers in an ISize object.

height Returns the height represented by the ISize object.

```
Coord height() const;
```

setHeight Sets the size's height.

```
ISize& setHeight( Coord cy);
```

setWidth Sets the size's width.

```
ISize& setWidth( Coord cx);
```

width Returns the width represented by the ISize object.

```
Coord width() const;
```

Inherited Public Functions

IPair		
asDebugInfo	operator !=	operator <=
asString	operator %=	operator ==
coord1	operator *=	operator >
coord2	operator +=	operator >=
distanceFrom	operator -	scaleBy
dotProduct	operator -=	scaledBy
maximum	operator /=	setCoord1
minimum	operator <	setCoord2

IBase		
asDebugInfo	**messageFile**	**setMessageFile**
asString	**messageText**	**version**

Inherited Protected Data

IBase		
recoverable	**unrecoverable**	

IStandardNotifier

Derivation

IBase
  IVBase
    INotifier
      IStandardNotifier

Inherited By

IMMDevice
IMMMasterAudio

Header File

istdntfy.hpp

Members

Member	Page	Member	Page
Constructor	464	notifyObservers	465
addObserver	465	observerList	465
deleteId	466	operator =	464
disableNotification	464	removeAllObservers	466
enableNotification	464	removeObserver	466
isEnabledForNotification	464	˜IStandardNotifier	464

The IStandardNotifier class provides a direct implementation of the notification protocol in the INotifier class.

You can implement a notification protocol in the following way:

- Derive a class from the IStandardNotifier class which inherits from INotifier for a direct implementation of the INotifier protocol
- Derive from the INotifier class and implement your own notification protocol

Because IWindow inherits from and implements the INotifier protocol, IWindow provides a visual notification implementation. IStandardNotifier inherits from INotifier and can be used for any generic notifier, without the visual interface available in IWindow objects. You might want to derive your classes from IStandardNotifier if you are providing a nonvisual notifier.

Public Functions

Constructors and Destructor

You can construct, destruct, assign, and copy objects of this class.

Constructors

1
```
IStandardNotifier( const IStandardNotifier& copy);
```

You can construct an IStandardNotifier object using a copy of an existing IStandardNotifier object.

2
```
IStandardNotifier();
```

You can construct objects of this class using the default constructor that takes no arguments.

operator = Assigns the contents of one notifier object to another.

Note: The observer list is not copied.

```
IStandardNotifier&
    operator =( const IStandardNotifier& aStandardNotifier);
```

Destructor
```
virtual ˜IStandardNotifier();
```

Notification Members

Use these members to affect the ability of a part to notify observers of events of interest.

disableNotification

Stops the object from sending notifications to registered observers.

```
virtual IStandardNotifier& disableNotification();
```

enableNotification

Starts the sending of notifications to observers.

```
virtual IStandardNotifier&
    enableNotification( Boolean enable = true);
```

isEnabledForNotification

Returns true if an object is sending notifications to its observers.

```
virtual Boolean isEnabledForNotification() const;
```

Observer Notification

These members notify observers of a change in a notifier.

notifyObservers

Notifies all observers in an object's observer list.

Note: A public and a protected version of notifyObservers are provided for convenience. The protected version does not require the caller to construct an INotificationEvent (p. 403) to call it. In this case, the construction of the INotificationEvent (p. 403) object occurs in the code of the protected notifyObservers function.

```
virtual IStandardNotifier&
   notifyObservers( const INotificationEvent& anEvent);
```

Inherited Public Functions

INotifier		
disableNotification	enableNotification	isEnabledForNotification

IVBase		
asDebugInfo	asString	

IBase		
asDebugInfo	**messageFile**	**setMessageFile**
asString	**messageText**	**version**

Protected Functions

Observer Addition and Removal

Use these members to manage the collection of observers maintained by the notifier.

addObserver Adds an observer to the object's list of observers.

```
virtual IStandardNotifier& addObserver( IObserver& observer,
   const IEventData& userData = IEventData ( 0 ));
```

observerList Returns the list of IObservers. The list is created it if it does not exist.

```
virtual IObserverList& observerList() const;
```

removeAllObservers

Removes all observers from the object's observer list.

```
virtual IStandardNotifier& removeAllObservers();
```

removeObserver

Removes an observer from the objects's observer list.

```
virtual IStandardNotifier& removeObserver( IObserver& observer);
```

Observer Notification

These members notify observers of a change in a notifier.

notifyObservers

Notifies all observers in an object's observer list.

Note: A public and a protected version of notifyObservers are provided for convenience. The protected version does not require the caller to construct an INotificationEvent (p. 403) to call it. In this case, the construction of the INotificationEvent (p. 403) object occurs in the code of the protected notifyObservers function.

```
virtual IStandardNotifier&
    notifyObservers( const INotificationId& nId);
```

Inherited Protected Functions

INotifier		
addObserver	notifyObservers	observerList

Public Data

Notification Event Descriptions

These INotificationId strings are used for all notifications that an IPart provides to its observers.

deleteId

Notification identifier provided to observers when the notifier object is deleted. Note: IStandardNotifier sends this notification from its destructor. This means that the derived portions of the notifier have already been deleted. You should therefore not cast the pointer to the notifier data, to an object that is derived from IStandardNotifier. This operation is synchronous and therefore the pointer still points to a valid object.

```
static INotificationId const deleteId;
```

Inherited Protected Data

IBase		
recoverable	**unrecoverable**	

IString

Derivation

IBase
  IString

Inherited By

I0String

Header File

istring.hpp

Members

Member	Page	Member	Page
Constructor	473	insert	477
applyBitOp	495	isAbbrevFor	498
asDebugInfo	475	isAbbreviationFor	487
asDouble	492	isAlphabetic	491
asInt	492	isAlphanumeric	491
asString	475	isASCII	491
asUnsigned	492	isBinaryDigits	491
b2c	472	isControl	491
b2d	472	isDBCS	485
b2x	472	isDigits	491
buffer	497	isGraphics	491
c2b	472	isHexDigits	491
c2d	472	isLike	487
c2x	473	isLowerCase	491
center	475	isMBCS	486
change	475	isPrintable	492
charType	488	isPunctuation	492
copy	477	isSBCS	486
d2b	486	isUpperCase	492
d2c	486	isValidDBCS	486
d2x	486	isValidMBCS	486
data	498	isWhiteSpace	492
defaultBuffer	498	lastIndexOf	489
findPhrase	497	lastIndexOfAnyBut	489
includes	487	lastIndexOfAnyOf	490
includesDBCS	485	leftJustify	477
includesMBCS	485	length	488
includesSBCS	485	lengthOf	498
indexOf	481	lengthOfWord	493
indexOfAnyBut	481	lineFrom	490
indexOfAnyOf	481	lowerCase	477
indexOfPhrase	493	maxLong	499
indexOfWord	493	null	499
initBuffer	497		

IString

Member	Page	Member	Page
nullBuffer	499	rightJustify	478
numWords	493	setBuffer	497
occurrencesOf	482	size	488
operator &	482	space	494
operator &=	483	strip	478
operator +	483	stripBlanks	479
operator +=	483	stripLeading	479
operator =	484	stripLeadingBlanks	479
operator char *	493	stripTrailing	479
operator signed char *	493	stripTrailingBlanks	480
operator unsigned char *	493	subString	488
operator []	488	translate	480
operator ^	484	upperCase	480
operator ^=	484	word	494
operator |	484	wordIndexOfPhrase	494
operator |=	485	words	494
operator ~	485	x2b	482
overlayWith	477	x2c	482
remove	478	x2d	482
removeWords	493	zero	499
reverse	478	~IString	475

Objects of the IString class are arrays of characters. These objects are functionally equivalent to objects of the class I0String (p. 307) with one major distinction: IStrings are indexed starting at 1 instead of 0.

IString provides an operator char*. In order to access the actual string contained in an object of type IString, cast the assignment variable implicitly or explicitly.

IStrings provide the following functions beyond that available from the standard C char* arrays and the STRING.H library functions:

- No restrictions on string contents. Thus, strings can contain NULL characters.
- Automatic conversion from and to numeric types.
- Automatic deletion of the string buffer when the IString is destroyed.
- Full support for the following:
  - All comparison operators
  - All bitwise operators
  - Concatenation using the more natural + operator.
- String data testing, such as for characters, digits, and hexadecimal digits.
- A full complement of the following:

  - String manipulation functions, such as center, left- and right-justification, stripping of leading and trailing characters, deleting substrings, and inserting strings
  - Corresponding string manipulation functions that return a new IString rather than modifying the receiver
  - String searching functions, such as byte index of string and last-byte index of string.
- Word manipulation, such as index of word and search for word phrase.
- Support for mixed strings that contain both single-byte character set (SBCS) and double-byte character set (DBCS) characters.

When a program using IStrings is run on a DBCS system, the IString objects support DBCS characters within the string contents. The various IString search functions do not accidentally match an SBCS character with the second byte of a DBCS character that has the same value. Also, IString functions that modify IStrings, such as subString (p. 488), remove (p. 478), and translate (p. 480), never separate the two bytes of a DBCS character. If one of the two bytes of a DBCS character is removed, the remaining byte is replaced with the appropriate pad character (if the function performing the change has one) or a blank.

When working with IStrings that contain DBCS data, ensure that the contents are not altered in such a way as to corrupt the data. For example, the statement:

```
aString[ n ] = 'x';
```

would be in error if the nth character of the IString was the first or second byte of a DBCS character.

Note: Any function that reallocates an IString can throw an exception for out-of-range errors. These occur if you attempt to construct an IString with a length greater than UINT_MAX.

IStrings are held in IBuffers which allocate the area for the character arrays using the C++ operator new. The only limitation for the size of an IString are the limitations imposed by the operating system.

Motif DBCS is equivalent to Multiple-byte character set (MBCS).

Public Functions

Binary Conversions

These members work if isBinaryDigits() == true; if not, they return a null string. The static members by the same name can be applied to a string to return the modified string without changing the argument string.

b2c Converts a string of binary digits to a normal string of characters. For example, this function changes 01 to \x01 and 00110011 to 3.

Note: This function is not locale sensitive.

```
IString& b2c();
static IString b2c( const IString& aString);
```

b2d Converts a string of binary digits to a string of decimal digits. For example, this function changes 00011001 to 25 and 0001001000110100 to 4660.

```
static IString b2d( const IString& aString);
IString& b2d();
```

b2x Converts a string of binary digits to a string of hexadecimal digits. For example, this function changes 00011011 to 1b and 10001001000110100 to 11234.

```
static IString b2x( const IString& aString);
IString& b2x();
```

Character Conversions

These members always work; they convert a string to binary, numeric or hexadecimal representation. The static members by the same name can be applied to a string to return the modified string without changing the argument string. These members are used much like the similar REXX functions. For example:

```
aString.c2b();                               // Changes aString.
String binaryDigits = IString::c2b( aString ); // Leaves aString alone.
```

c2b Converts a normal string of characters to a string of binary digits. For example, this function changes “a” to 01100001 and 12 to 11000100110010.

Note: This function is not locale-sensitive.

```
IString& c2b();
static IString c2b( const IString& aString);
```

c2d Converts a normal string of characters to a string of decimal digits. For example, this function changes “a” to 97 and “ab” to 24930.

Note: This function is not locale sensitive.

```
static IString c2d( const IString& aString);
IString& c2d();
```

c2x Converts a normal string of characters to a string of hexadecimal digits. For example, this function changes 'a' to 61 and 'ab' to 6162.

Note: This function is not locale sensitive.

```
static IString c2x( const IString& aString);
IString& c2x();
```

Constructors and Destructor

You can construct objects of this class in the following ways:

- Construct a NULL string.
- Construct a string with the ASCII representation of a given numeric value, supporting all flavors of integer and double.
- Construct a string with a copy of the specified character data, supporting ASCIIZ strings, characters, and IStrings. The character data passed is converted to its ASCII representation.
- Construct a string with contents that consist of copies of up to three buffers of arbitrary data (void*). Optionally, you only need to provide the length, in which case the IString contents are initialized to a specified pad character. The default character is a blank.

These constructors can throw exceptions under the following conditions:

- Memory allocation errors

  Many factors dynamically allocate space and these allocation requests may fail. If so, the library translates memory allocation errors into exceptions. Generally, such errors do not occur until you allocate an astronomical amount of storage.

- Out-of-range errors

  These occur if you attempt to construct an IString with a length greater than UINT_MAX.

Constructors

1
```
IString( const void* pBuffer1, unsigned lenBuffer1,
    char padCharacter = ' ');
```

Construct a string with contents from one buffer of arbitrary data (void*).

2
```
IString();
```

Construct a NULL string.

3
```
IString( const IString& aString);
```

Construct a string with a copy of the specified IString.

4
```
IString( int);
```

Construct a string with the ASCII representation of an integer value.

5
```
IString( unsigned);
```

Construct a string with the ASCII representation of an unsigned numeric value.

6
```
IString( long);
```

Construct a string with the ASCII representation of a long numeric value.

7
```
IString( unsigned long);
```

Construct a string with the ASCII representation of an unsigned long numeric value.

8
```
IString( short);
```

Construct a string with the ASCII representation of a short numeric value.

9
```
IString( unsigned short);
```

Construct a string with the ASCII representation of an unsigned short numeric value.

10
```
IString( double);
```

Construct a string with the ASCII representation of a double numeric value.

11
```
IString( char);
```

Construct a string with a copy of the character. The string length is set to 1.

12
```
IString( unsigned char);
```

Construct a string with a copy of the unsigned character. The string length is set to 1.

13
```
IString( signed char);
```

Construct a string with a copy of the signed character. The string length is set to 1.

14
```
IString( const char*);
```

Construct a string with a copy of the specified ASCIIZ string.

15
```
IString( const unsigned char*);
```

Construct a string with a copy of the specified unsigned ASCIIZ string.

16
```
IString( const signed char*);
```

Construct a string with a copy of the specified signed ASCIIZ string.

17
```
IString( const void* pBuffer1, unsigned lenBuffer1,
    const void* pBuffer2, unsigned lenBuffer2,
    char padCharacter = ' ');
```

Construct a string with contents from two buffers of arbitrary data (void*).

18
```
IString( const void* pBuffer1, unsigned lenBuffer1,
    const void* pBuffer2, unsigned lenBuffer2,
    const void* pBuffer3, unsigned lenBuffer3,
    char padCharacter = ' ');
```

Construct a string with contents from three buffers of arbitrary data (void*).

Destructor
```
~IString();
```

Diagnostics

These members provide IString diagnostic information for IString objects. Often, you use these members to write trace information when debugging.

asDebugInfo Returns information about the IString's internal representation that you can use for debugging.

```
IString asDebugInfo() const;
```

asString Returns the string itself, so that IString supports this common IBase (p. 323) protocol.

```
IString asString() const;
```

Editing

Use these members to edit a string. All return a reference to the modified receiver. Many that are length related, such as center and leftJustify, accept a pad character that defaults to a blank. In all cases, you can specify argument strings as either objects of the IString class or by using char*.

Static members by the same name can be applied to an IString to obtain the modified IString without affecting the argument. For example:

```
aString.change('\t', '   ');                  // Changes all tabs in aString to 3 blanks.
IString s = IString::change( aString, '\t', '   ' );  // Leaves aString as is.
```

center Centers the receiver within a string of the specified length.

```
IString& center( unsigned length, char padCharacter = ' ');
static IString center( const IString& aString,
    unsigned length, char padCharacter = ' ');
```

change Changes occurrences of a specified pattern to a specified replacement string. You can specify the number of changes to perform. The default is to change all occurrences of the pattern. You can also specify the position in the receiver at which to begin.

The parameters are the following:

inputString
The pattern string as a reference to an object of type IString. The library searches for the pattern string within the receiver's data.

pInputString
The pattern string as NULL-terminated string. The library searches for the pattern string within the receiver's data.

outputString
The replacement string as a reference to an object of type IString. It replaces the occurrences of the pattern string in the receiver's data.

pOutputString
The replacement string as a NULL-terminated string. It replaces the occurrences of the pattern string in the receiver's data.

startPos The position to start the search at within the receiver's data. The default is 1.

numChanges
The number of patterns to search for and change. The default is UINT_MAX, which causes changes to all occurrences of the pattern.

```
static IString change( const IString& aString,
    const char* pInputString, const char* pOutputString,
    unsigned startPos = 1,
    unsigned numChanges = ( unsigned ) UINT_MAX);

IString& change( const IString& inputString,
    const IString& outputString, unsigned startPos = 1,
    unsigned numChanges = ( unsigned ) UINT_MAX);

IString& change( const IString& inputString,
    const char* pOutputString, unsigned startPos = 1,
    unsigned numChanges = ( unsigned ) UINT_MAX);

IString& change( const char* pInputString,
    const IString& outputString, unsigned startPos = 1,
    unsigned numChanges = ( unsigned ) UINT_MAX);

IString& change( const char* pInputString,
    const char* pOutputString, unsigned startPos = 1,
    unsigned numChanges = ( unsigned ) UINT_MAX);

static IString change( const IString& aString,
    const IString& inputString,
    const IString& outputString,
    unsigned startPos = 1,
    unsigned numChanges = ( unsigned ) UINT_MAX);
```

```
static IString change( const IString& aString,
    const IString& inputString, const char* pOutputString,
    unsigned startPos = 1, unsigned numChanges = ( unsigned ) UINT_MAX);

static IString change( const IString& aString,
    const char* pInputString, const IString& outputString,
    unsigned startPos = 1, unsigned numChanges = ( unsigned ) UINT_MAX);
```

copy Replaces the receiver's contents with a specified number of replications of itself.

```
static IString copy( const IString& aString, unsigned numCopies);
IString& copy( unsigned numCopies);
```

insert Inserts the specified string after the specified location.

```
static IString insert( const IString& aString,
    const char* pInsert, unsigned index = 0,
    char padCharacter = ' ');

IString& insert( const IString& aString,
    unsigned index = 0, char padCharacter = ' ');

IString& insert( const char* pString,
    unsigned index = 0, char padCharacter = ' ');

static IString insert( const IString& aString,
    const IString& anInsert, unsigned index = 0,
    char padCharacter = ' ');
```

leftJustify Left-justifies the receiver in a string of the specified length. If the new length (*length*) is larger than the current length, the string is extended by the pad character (*padCharacter*). The default pad character is a blank.

```
static IString leftJustify( const IString& aString,
    unsigned length, char padCharacter = ' ');

IString& leftJustify( unsigned length, char padCharacter = ' ');
```

lowerCase Translates all upper-case letters in the receiver to lower-case.

```
static IString lowerCase( const IString& aString);
IString& lowerCase();
```

overlayWith Replaces a specified portion of the receiver's contents with the specified string. The overlay starts in the receiver's data at the *index*, which defaults to 1. If *index* is beyond the end of the receiver's data, it is padded with the pad character (*padCharacter*).

```
static IString overlayWith( const IString& aString,
    const IString& anOverlay, unsigned index = 1,
    char padCharacter = ' ');

IString& overlayWith( const IString& aString,
    unsigned index = 1, char padCharacter = ' ');

IString& overlayWith( const char* pString,
    unsigned index = 1, char padCharacter = ' ');

static IString overlayWith( const IString& aString,
    const char* pOverlay, unsigned index = 1,
    char padCharacter = ' ');
```

remove

Deletes the specified portion of the string (that is, the substring) from the receiver. You can use this function to truncate an IString object at a specific position. For example:

```
aString.remove(8);
```

removes the substring beginning at index 8 and takes the rest of the string as a default.

```
IString& remove( unsigned startPos, unsigned numChars);
IString& remove( unsigned startPos);
static IString remove( const IString& aString, unsigned startPos);
static IString remove( const IString& aString,
    unsigned startPos, unsigned numChars);
```

reverse

Reverses the receiver's contents.

```
IString& reverse();
static IString reverse( const IString& aString);
```

rightJustify

Right-justifies the receiver in a string of the specified length. If the receiver's data is shorter than the requested length (*length*), it is padded on the left with the pad character (*padCharacter*). The default pad character is a blank.

```
static IString rightJustify( const IString& aString,
    unsigned length, char padCharacter = ' ');

IString& rightJustify( unsigned length, char padCharacter = ' ');
```

strip

Strips both leading and trailing character or characters. You can specify the character or characters as the following:

- A single char
- A char* array
- An IString (p. 469) object
- An IStringTest (p. 515) object

The default is white space.

```
IString& strip( const char* pString);
IString& strip();
IString& strip( char aCharacter);
IString& strip( const IString& aString);
IString& strip( const IStringTest& aTest);
static IString strip( const IString& aString, char aChar);
static IString strip( const IString& aString,
    const IString& aStringOfChars);
static IString strip( const IString& aString,
    const char* pStringOfChars);
static IString strip( const IString& aString,
    const IStringTest& aTest);
```

stripBlanks Strips both leading and trailing white space.

Note: This function is the static version of IString::strip (p. 478), which has been renamed to avoid a duplicate definition.

```
static IString stripBlanks( const IString& aString);
```

stripLeading Strips the leading character or characters.

```
static IString stripLeading( const IString& aString, char aChar);
IString& stripLeading();
IString& stripLeading( char aCharacter);
IString& stripLeading( const IString& aString);
IString& stripLeading( const char* pString);
IString& stripLeading( const IStringTest& aTest);
static IString stripLeading( const IString& aString,
    const IString& aStringOfChars);
static IString stripLeading( const IString& aString,
    const char* pStringOfChars);
static IString stripLeading( const IString& aString,
    const IStringTest& aTest);
```

stripLeadingBlanks

Strips the leading character or characters.

Note: This function is the static version of IString::stripLeading (p. 479), which has been renamed to avoid a duplicate definition.

```
static IString stripLeadingBlanks( const IString& aString);
```

stripTrailing Strips the trailing character or characters.

```
static IString stripTrailing( const IString& aString,
    const IStringTest& aTest);
IString& stripTrailing();
IString& stripTrailing( char aCharacter);
IString& stripTrailing( const IString& aString);
IString& stripTrailing( const char* pString);
```

```
IString& stripTrailing( const IStringTest& aTest);
static IString stripTrailing( const IString& aString,
    char aChar);
static IString stripTrailing( const IString& aString,
    const IString& aStringOfChars);
static IString stripTrailing( const IString& aString,
    const char* pStringOfChars);
```

stripTrailingBlanks

Strips the trailing character or characters.

Note: This function is the static version of IString::stripTrailing (p. 479), which has been renamed to avoid a duplicate definition.

```
static IString stripTrailingBlanks( const IString& aString);
```

translate

Converts all of the receiver's characters that are in the first specified string to the corresponding character in the second specified string.

```
static IString translate( const IString& aString,
    const char* pInputChars, const IString& outputChars,
    char padCharacter = ' ');

IString& translate( const IString& inputChars,
    const IString& outputChars, char padCharacter = ' ');

IString& translate( const IString& inputChars,
    const char* pOutputChars, char padCharacter = ' ');

IString& translate( const char* pInputChars,
    const IString& outputChars, char padCharacter = ' ');

IString& translate( const char* pInputChars,
    const char* pOutputChars, char padCharacter = ' ');

static IString translate( const IString& aString,
    const IString& inputChars, const IString& outputChars,
    char padCharacter = ' ');

static IString translate( const IString& aString,
    const IString& inputChars, const char* pOutputChars,
    char padCharacter = ' ');

static IString translate( const IString& aString,
    const char* pInputChars, const char* pOutputChars,
    char padCharacter = ' ');
```

upperCase

Translates all lower-case letters in the receiver to upper-case.

```
IString& upperCase();
static IString upperCase( const IString& aString);
```

Forward Searches

These members permit searching a string in various ways. You can specify an optional index that indicates the search start position. The default starts at the beginning of the string.

indexOf Returns the byte index of the first occurrence of the specified string within the receiver. If there are no occurrences, 0 is returned. In addition to IStrings, you can also specify a single character or an IStringTest (p. 515).

```
unsigned indexOf( const IString& aString,
    unsigned startPos = 1) const;
unsigned indexOf( const char* pString, unsigned startPos = 1) const;
unsigned indexOf( char aCharacter, unsigned startPos = 1) const;
unsigned indexOf( const IStringTest& aTest,
    unsigned startPos = 1) const;
```

indexOfAnyBut

Returns the index of the first character of the receiver that is not in the specified set of characters. If there are no characters, 0 is returned. Alternatively, this function returns the index of the first character that fails the test prescribed by a specified IStringTest (p. 515) object.

```
unsigned indexOfAnyBut( const IString& validChars,
    unsigned startPos = 1) const;

unsigned indexOfAnyBut( const char* pValidChars,
    unsigned startPos = 1) const;

unsigned indexOfAnyBut( char validChar, unsigned startPos = 1) const;

unsigned indexOfAnyBut( const IStringTest& aTest,
    unsigned startPos = 1) const;
```

indexOfAnyOf

Returns the index of the first character of the receiver that is a character in the specified set of characters. If there are no characters, 0 is returned. Alternatively, this function returns the index of the first character that passes the test prescribed by a specified IStringTest (p. 515) object.

```
unsigned indexOfAnyOf( const char* pSearchChars,
    unsigned startPos = 1) const;

unsigned indexOfAnyOf( const IString& searchChars,
    unsigned startPos = 1) const;

unsigned indexOfAnyOf( char searchChar, unsigned startPos = 1) const;

unsigned indexOfAnyOf( const IStringTest& aTest,
    unsigned startPos = 1) const;
```

occurrencesOf

Returns the number of occurrences of the specified IString, char*, char, or IStringTest. If you just want a Boolean test, this is slower than IString::indexOf (p. 481).

```
unsigned occurrencesOf( const IStringTest& aTest,
    unsigned startPos = 1) const;

unsigned occurrencesOf( const IString& aString,
    unsigned startPos = 1) const;

unsigned occurrencesOf( const char* pString,
    unsigned startPos = 1) const;

unsigned occurrencesOf( char aCharacter, unsigned startPos = 1) const;
```

Hex Conversions

These members work if isHexDigits() == true; if not, they return a null string. The static members by the same name can be applied to a string to return the modified string without changing the argument string.

x2b

Converts a string of hexadecimal digits to a string of binary digits. For example, this function changes a1c to 101000011100 and f3 to 11110011.

```
IString& x2b();
static IString x2b( const IString& aString);
```

x2c

Converts a string of hexadecimal digits to a normal string of characters. For example, this function changes 8 to \x08 and 31393935 to 1995.

Note: This function is not locale sensitive.

```
static IString x2c( const IString& aString);
IString& x2c();
```

x2d

Converts a string of hexadecimal digits to a string of decimal digits. For example, this function changes a1c to 2588 and 10000 to 65536.

```
static IString x2d( const IString& aString);
IString& x2d();
```

Manipulation

Use these members to manipulate a string's contents. All are overloaded so that standard C strings can be used efficiently without constructing an equivalent String first.

operator &

Performs bitwise AND. This function can handle the following three forms:

string1 & aString
Both operands are of type IString.

string1 & pString
The first operand is an IString and the second is a NULL-terminated character string.

pString & aString
The first operand is a NULL-terminated character string and the second is an IString.

```
IString operator &( const char* pString) const;
IString operator &( const IString& aString) const;
```

operator &= Performs bitwise AND and replaces the receiver. This function can handle the following two forms:

string1 &= aString
Both operands are of type IString.

string1 &= pString
The first operand is an IString and the second is a NULL-terminated character string.

```
IString& operator &=( const char* pString);
IString& operator &=( const IString& aString);
```

operator + Concatenates two strings. This function can handle the following three forms:

string1 + aString
Both operands are of type IString.

string1 + pString
The first operand is an IString and the second is a NULL-terminated character string.

pString + aString
The first operand is a NULL-terminated character string and the second is an IString.

```
IString operator +( const IString& aString) const;
IString operator +( const char* pString) const;
```

operator += Concatenates the specified string to the receiver and replaces the receiver. This function can handle the following two forms:

string1 += aString
Both operands are of type IString.

string1 += pString
The first operand is an IString and the second is a NULL-terminated character string.

```
IString& operator +=( const char* pString);
IString& operator +=( const IString& aString);
```

operator = Replaces the contents of the string.

```
IString& operator =( const IString& aString);
```

operator ^

Performs bitwise XOR. This function can handle the following three forms:

string1 ^ aString
Both operands are of type IString.

string1 ^ pString
The first operand is an IString and the second is a NULL-terminated character string.

pString ^ aString
The first operand is a NULL-terminated character string and the second is an IString.

```
IString operator ^( const char* pString) const;
IString operator ^( const IString& aString) const;
```

operator ^=

Performs bitwise XOR and replaces the receiver. This function can handle the following two forms:

string1 ^= aString
Both operands are of type IString.

string1 ^= pString
The first operand is an IString and the second is a NULL-terminated character string.

```
IString& operator ^=( const IString& aString);
IString& operator ^=( const char* pString);
```

operator | Performs bitwise OR. This function can handle the following three forms:

string1 | aString
Both operands are of type IString.

string1 | pString
The first operand is an IString and the second is a NULL-terminated character string.

pString | aString
The first operand is a NULL-terminated character string and the second is an IString.

```
IString operator |( const char* pString) const;
IString operator |( const IString& aString) const;
```

operator |=

Performs bitwise OR and replaces the receiver with the resulting string. This function can handle the following two forms:

string1 |= aString
Both operands are of type IString.

string1 |= pString
The first operand is an IString and the second is a NULL-terminated character string.

```
IString& operator |=( const IString& aString);
IString& operator |=( const char* pString);
```

operator ˜ Returns the string's bitwise negation (the string's complement).

```
IString operator ˜() const;
```

NLS Testing

Use these members to test the characters that comprise a string. Basically, you use these members to determine if an IString contains only characters from a specific NLS character set (SBCS, MBCS, DBCS).

includesDBCS

If any characters are DBCS (double-byte character set), true is returned.

Note: This function is interchangeable with includesMBCS.

```
Boolean includesDBCS() const;
```

includesMBCS

If any characters are MBCS (multiple-byte character set), true is returned.

Note: This function is interchangeable with includesDBCS.

```
Boolean includesMBCS() const;
```

includesSBCS

If any characters are SBCS (single-byte character set), true is returned.

```
Boolean includesSBCS() const;
```

isDBCS If all the characters are DBCS, true is returned.

Note: This function is interchangeable with isMBCS.

```
Boolean isDBCS() const;
```

isMBCS If all the characters are MBCS, true is returned.

Note: This function is interchangeable with isDBCS.

```
Boolean isMBCS() const;
```

isSBCS If all the characters are SBCS, true is returned.

```
Boolean isSBCS() const;
```

isValidDBCS If no DBCS characters have a 0 second byte, true is returned.

Note: This function is interchangeable with isValidMBCS.

```
Boolean isValidDBCS() const;
```

isValidMBCS If no MBCS characters have a 0 second byte, true is returned.

Note: This function is interchangeable with isValidDBCS.

```
Boolean isValidMBCS() const;
```

Numeric Conversions

These members work if isDigits() == true; if not, they return a null string. The static members by the same name can be applied to a string to return the modified string without changing the argument string.

d2b Converts a string of decimal digits to a string of binary digits. This function builds the string eight bits at a time. For example,

```
  '12' gets converted to '00001100'
  '17' gets converted to '00010001'
  '123' gets converted to '01111011'
```

Use stripLeading('0') to strip the leading zeros.

```
IString& d2b();
static IString d2b( const IString& aString);
```

d2c Converts a string of decimal digits to a normal string of characters. For example, this function changes 12 to \x0c and 56 to 8.

Note: This function is not locale sensitive.

```
static IString d2c( const IString& aString);
IString& d2c();
```

d2x Converts a string of decimal digits to a string of hexadecimal digits. For example, this function changes 12 to c and 123 to 7b.

```
static IString d2x( const IString& aString);
IString& d2x();
```

Pattern Matching

Use these members to determine if an object of this class contains a given pattern of characters.

includes

If the receiver contains the specified search string, true is returned.

```
Boolean includes( const IStringTest& aTest) const;
Boolean includes( const IString& aString) const;
Boolean includes( const char* pString) const;
Boolean includes( char aChar) const;
```

isAbbreviationFor

If the receiver is a valid abbreviation of the specified string, true is returned.

The parameters are the following:

fullString The full string for the abbreviation check is contained in another IString.

pFullString
The full string for the abbreviation check is a NULL-terminated character string.

minAbbrevLength
The minimum length to match for it to be a valid abbreviation. The default minimum length is 0, which means the minimum length is the length of the receiver's string.

```
Boolean isAbbreviationFor( const char* pFullString,
    unsigned minAbbrevLength = 0) const;

Boolean isAbbreviationFor( const IString& fullString,
    unsigned minAbbrevLength = 0) const;
```

isLike

If the receiver matches the specified pattern, which can contain wildcard characters, true is returned.

- You can use the first wildcard character to specify that 0 or more arbitrary characters are accepted. The default wildcard character that does this is *, but you can specify another character when calling IString::isLike. For example:

  ```
  IString( "Allison" ).isLike( "Al*ison" ) -> true
  ```

- You can use the second wildcard character to specify that a single arbitrary character is accepted. The default wildcard character that does this is ?, but you can specify another character when calling IString::isLike. For example:

  ```
  IString( "istring7.cpp" ).isLike( "i*.?pp" ) -> true
  IString( "Not a question!" ).isLike( "*?", '*', '-' ) -> false
  ```

```
Boolean isLike( const char* pPattern,
    char zeroOrMore = ' * ', char anyChar = '?') const;

Boolean isLike( const IString& aPattern,
    char zeroOrMore = ' * ', char anyChar = '?') const;
```

Queries

Use these members to access general information about the string.

charType Returns the type of the character at the specified index.

```
IStringEnum::CharType charType( unsigned index) const;
```

length Returns the length of the string, not counting the terminating NULL character.

```
unsigned length() const;
```

operator [] Returns a reference to the specified character of the string.

Note: If you call the non-const version of this function with an index beyond the end, the function extends the string.

1
```
const char& operator []( unsigned index) const;
```

Exception

IInvalidRequest. Passed an index larger than the string size. Possible causes include boundary errors and using this function instead of the non-const version which grows the underlying IString buffer to accommodate the index value.

2
```
char& operator []( unsigned index);
```

size Returns the length of the string, not counting the terminating NULL character.

```
unsigned size() const;
```

subString Returns a specified portion of the string (that is, the substring) of the receiver.

The parameters are the following:

startPos The starting position of the substring being extracted. If this position is beyond the end of the data in the receiver, this function returns a NULL IString.

length The length of the substring to be extracted. If the length extends beyond the end of the receiver's data, the returned IString is padded to the specified length with *padCharacter*. If you do not specify *length* and it defaults, this function uses the rest of the receiver's data starting from *startPos* for padding.

padCharacter
: The character the function uses as padding if the requested length extends beyond the end of the receiver's data. The default *padCharacter* is a blank.

You can use this function to truncate an IString object at a specific position. For example:

```
aString = aString.subString(1, 7);
```

returns the substring concluding with index 7 and discards the rest of the string.

```
IString subString( unsigned startPos) const;
IString subString( unsigned startPos,
    unsigned length, char padCharacter = ' ') const;
```

Reverse Searches

These members permit searching the string in various ways. The lastIndexOf versions correspond to forward search indexOf members but start the search from the end of the string. These members return the index of the last character in the receiver IString that satisfies the search criteria. Also, they accept an optional argument that specifies where the search is to begin. The default is to start searching at the end of the string. Searching proceeds from right to left for these members.

lastIndexOf Returns the index of the last occurrence of the specified string or character. The search starts at the position specified by *startPos* (inclusive) and proceeds backward. The returned value is in the range *0 <= x <= startPos*. The default of UINT_MAX starts the search at the end of the receiver's string. If the search target is not found, 0 is returned.

If you specify 1 for *startPos*, the search starts at the beginning of the string. Therefore, because the search proceeds backward from its starting position, in this case the search target must occur at the beginning of the string for it to be found.

```
unsigned lastIndexOf( const char* pString,
    unsigned startPos = ( unsigned ) UINT_MAX) const;

unsigned lastIndexOf( const IString& aString,
    unsigned startPos = ( unsigned ) UINT_MAX) const;

unsigned lastIndexOf( char aCharacter,
    unsigned startPos = ( unsigned ) UINT_MAX) const;

unsigned lastIndexOf( const IStringTest& aTest,
    unsigned startPos = ( unsigned ) UINT_MAX) const;
```

lastIndexOfAnyBut

Returns the index of the last character not in the specified string or character. The search starts at the position specified by *startPos* (inclusive) and proceeds backward.

The default of UINT_MAX starts the search at the end of the receiver's string. If the search target is not found, 0 is returned.

If you specify 1 for *startPos*, the search starts at the beginning of the string. Therefore, because the search proceeds backward from its starting position, in this case the search target must occur at the beginning of the string for it to be found.

```
unsigned lastIndexOfAnyBut( const IString& validChars,
    unsigned startPos = ( unsigned ) UINT_MAX) const;

unsigned lastIndexOfAnyBut( const char* pValidChars,
    unsigned startPos = ( unsigned ) UINT_MAX) const;

unsigned lastIndexOfAnyBut( char validChar,
    unsigned startPos = ( unsigned ) UINT_MAX) const;

unsigned lastIndexOfAnyBut( const IStringTest& aTest,
    unsigned startPos = ( unsigned ) UINT_MAX) const;
```

lastIndexOfAnyOf
Returns the index of the last character in the specified string or character. The search starts at the position specified by *startPos* (inclusive) and proceeds backward. The default of UINT_MAX starts the search at the end of the receiver's string. If the search target is not found, 0 is returned.

If you specify 1 for *startPos*, the search starts at the beginning of the string. Therefore, because the search proceeds backward from its starting position, in this case the search target must occur at the beginning of the string for it to be found.

```
unsigned lastIndexOfAnyOf( const char* pSearchChars,
    unsigned startPos = ( unsigned ) UINT_MAX) const;

unsigned lastIndexOfAnyOf( const IString& searchChars,
    unsigned startPos = ( unsigned ) UINT_MAX) const;

unsigned lastIndexOfAnyOf( char searchChar,
    unsigned startPos = ( unsigned ) UINT_MAX) const;

unsigned lastIndexOfAnyOf( const IStringTest& aTest,
    unsigned startPos = ( unsigned ) UINT_MAX) const;
```

Stream Input

Use these members to read IStrings from standard C++ streams.

lineFrom Returns the next line from the specified input stream. This static function accepts an optional line delimiter, which defaults to \n. The resulting IString contains the characters up to the next occurrence of the delimiter. The delimiter character is skipped. If an EOF condition occurs, this function returns an IString whose contents are NULL.

```
static IString lineFrom( istream& aStream, char delim = '\n');
```

Testing

Use these members to determine if an IString contains only characters from a predefined set.

isAlphabetic If all the characters are in {'A'-'Z','a'-'z'}, true is returned.

```
Boolean isAlphabetic() const;
```

isAlphanumeric
If all the characters are in {'A'-'Z','a'-'z','0'-'9'}, true is returned.

```
Boolean isAlphanumeric() const;
```

isASCII If all the characters are in {0x00-0x7F}, true is returned.

```
Boolean isASCII() const;
```

isBinaryDigits
If all the characters are either 0 or 1, true is returned.

```
Boolean isBinaryDigits() const;
```

isControl Returns true if all the characters are control characters. Control characters are determined using the iscntrl() C Library function defined in the cntrl locale source file and in the cntrl class of the LC_CTYPE category of the current locale. For example, on ASCII operating systems, control characters are those in the range {0x00-0x1F,0x7F}.

```
Boolean isControl() const;
```

isDigits If all the characters are in {'0'-'9'}, true is returned.

```
Boolean isDigits() const;
```

isGraphics Returns true if all the characters are graphics characters.

Graphics characters are printable characters excluding the space character, as defined by the isgraph() C Library function in the graph locale source file and in the graph class of the LC_CTYPE category of the current locale. On ASCII systems, for example, graphics characters are those in the range {0x21-0x7E}.

```
Boolean isGraphics() const;
```

isHexDigits If all the characters are in {'0'-'9','A'-'F','a'-'f'}, true is returned.

```
Boolean isHexDigits() const;
```

isLowerCase If all the characters are in {'a'-'z'}, true is returned.

```
Boolean isLowerCase() const;
```

isPrintable Returns true if all the characters are printable characters. Printable characters are defined by the isprint() C Library function as defined in the print locale source file and in the print class of the LC_CTYPE category of the current locale. On ASCII systems, for example, printable characters are those in the range {0x20-0x7E}.

```
Boolean isPrintable() const;
```

isPunctuation

If none of the characters is white space, a control character, or an alphanumeric character, true is returned.

```
Boolean isPunctuation() const;
```

isUpperCase If all the characters are in {'A'-'Z'}, true is returned.

```
Boolean isUpperCase() const;
```

isWhiteSpace

Returns true if all the characters are whitespace characters. Whitespace characters are defined by the isspace() C Library function as defined in the space locale source file and in the space class of the LC_CTYPE category of the current locale. For example, on ASCII systems, whitespace characters are those in the range {0x09-0x0D,0x20}.

```
Boolean isWhiteSpace() const;
```

Type Conversions

Use these members to convert a string to various other data types. The types supported are the same set as are supported by the IString constructors.

asDouble Returns, as a double, the number that the string represents.

```
double asDouble() const;
```

asInt Returns the number that the string represents as a long integer.

Note: If an IString contains nonnumeric characters, this function returns the integer for the portion of the IString up to, but not including, the nonnumeric character. The rest of the IString, following the invalid character, is not returned.

If an IString is larger than the maximum integer, this function returns the maximum integer, not the larger value.

```
long asInt() const;
```

asUnsigned Returns, as an unsigned long, the integer that the string represents.

```
unsigned long asUnsigned() const;
```

operator char *

Returns a char* pointer to the string's contents.

```
operator char *() const;
```

operator signed char *

Returns a signed char* pointer to the string's contents.

```
operator signed char *() const;
```

operator unsigned char *

Returns an unsigned char* pointer to the string's contents.

```
operator unsigned char *() const;
```

Word Operations

These members operate on a string as a collection of words separated by whitespace characters. They find, remove, and count words or phrases.

indexOfPhrase

Returns the position of the first occurrence of the specified phrase in the receiver. If the phrase is not found, 0 is returned.

```
unsigned indexOfPhrase( const IString& wordString,
    unsigned startWord = 1) const;
```

indexOfWord

Returns the index of the specified white-space-delimited word in the receiver. If the word is not found, 0 is returned.

```
unsigned indexOfWord( unsigned wordNumber) const;
```

lengthOfWord

Returns the length of the specified white-space-delimited word in the receiver.

```
unsigned lengthOfWord( unsigned wordNumber) const;
```

numWords

Returns the number of words in the receiver.

```
unsigned numWords() const;
```

removeWords

Deletes the specified words from the receiver's contents. You can specify the words by using a starting word number and the number of words. The latter defaults to the rest of the string.

Note: The static functions IString::space (p. 494) and IString::removeWords obtain the same result but do not affect the String to which they are applied.

```
IString& removeWords( unsigned firstWord);

IString& removeWords( unsigned firstWord, unsigned numWords);

static IString removeWords( const IString& aString,
    unsigned startWord);

static IString removeWords( const IString& aString,
    unsigned startWord, unsigned numWords);
```

space

Modifies the receiver so that all words are separated by the specified number of blanks. The default is one blank. All white space is converted to simple blanks.

Note: The static functions IString::space and IString::removeWords (p. 493) obtain the same result but do not affect the String to which they are applied.

```
static IString space( const IString& aString,
    unsigned numSpaces = 1, char spaceChar = ' ');

IString& space( unsigned numSpaces = 1, char spaceChar = ' ');
```

word

Returns a copy of the specified white-space-delimited word in the receiver.

```
IString word( unsigned wordNumber) const;
```

wordIndexOfPhrase

Returns the word number of the first word in the receiver that matches the specified phrase. The function starts its search with the word number you specify in *startWord*, which defaults to 1. If the phrase is not found, 0 is returned.

```
unsigned wordIndexOfPhrase( const IString& aPhrase,
    unsigned startWord = 1) const;
```

words

Returns a substring of the receiver that starts at a specified word and is comprised of a specified number of words. The word separators are copied to the result intact.

```
IString words( unsigned firstWord, unsigned numWords) const;
IString words( unsigned firstWord) const;
```

Inherited Public Functions

IBase		
asDebugInfo	**messageFile**	**setMessageFile**
asString	**messageText**	**version**

Protected Functions

Bit Operations

Use these members to implement various public members of this class requiring bitwise operations.

applyBitOp Implements the bitwise operators &, |, and ^.

```
IString& applyBitOp( const char* pArg,
    unsigned argLen, BitOperator op);
```

Editing

Use these members to edit a string. All return a reference to the modified receiver. Many that are length related, such as center and leftJustify, accept a pad character that defaults to a blank. In all cases, you can specify argument strings as either objects of the IString class or by using char*.

Static members by the same name can be applied to an IString to obtain the modified IString without affecting the argument. For example:

```
aString.change('\t', '   ');                  // Changes all tabs in aString to 3 blanks.
IString s = IString::change( aString, '\t', '   ' );  // Leaves aString as is.
```

change Changes occurrences of a specified pattern to a specified replacement string. You can specify the number of changes to perform. The default is to change all occurrences of the pattern. You can also specify the position in the receiver at which to begin.

The parameters are the following:

inputString
The pattern string as a reference to an object of type IString. The library searches for the pattern string within the receiver's data.

pInputString
The pattern string as NULL-terminated string. The library searches for the pattern string within the receiver's data.

outputString
The replacement string as a reference to an object of type IString. It replaces the occurrences of the pattern string in the receiver's data.

pOutputString
The replacement string as a NULL-terminated string. It replaces the occurrences of the pattern string in the receiver's data.

startPos The position to start the search at within the receiver's data. The default is 1.

numChanges
The number of patterns to search for and change. The default is UINT_MAX, which causes changes to all occurrences of the pattern.

```
IString& change( const char* pPattern, unsigned patternLen,
    const char* pReplacement, unsigned replacementLen,
    unsigned startPos, unsigned numChanges);
```

insert
Inserts the specified string after the specified location.

```
IString& insert( const char* pInsert, unsigned insertLen,
    unsigned startPos, char padCharacter);
```

overlayWith
Replaces a specified portion of the receiver's contents with the specified string. The overlay starts in the receiver's data at the *index*, which defaults to 1. If *index* is beyond the end of the receiver's data, it is padded with the pad character (*padCharacter*).

```
IString& overlayWith( const char* pOverlay,
    unsigned overlayLen, unsigned index,
    char padCharacter);
```

strip
Strips both leading and trailing character or characters. You can specify the character or characters as the following:

- A single char
- A char* array
- An IString (p. 469) object
- An IStringTest (p. 515) object

The default is white space.

```
IString& strip( const char* p, unsigned len,
    IStringEnum::StripMode mode);

IString& strip( const IStringTest& aTest,
    IStringEnum::StripMode mode);
```

translate
Converts all of the receiver's characters that are in the first specified string to the corresponding character in the second specified string.

```
IString& translate( const char* pInputChars,
    unsigned inputLen, const char* pOutputChars,
    unsigned outputLen, char padCharacter);
```

Forward Searches

These members permit searching a string in various ways. You can specify an optional index that indicates the search start position. The default starts at the beginning of the string.

findPhrase Locates a specified string of words for indexOfWord functions.

```
unsigned findPhrase( const IString& aPhrase,
    unsigned startWord, IndexType charOrWord) const;
```

indexOfWord Returns the index of the specified white-space-delimited word in the receiver. If the word is not found, 0 is returned.

```
unsigned indexOfWord( unsigned wordNumber,
    unsigned startPos, unsigned numWords) const;
```

occurrencesOf Returns the number of occurrences of the specified IString, char*, char, or IStringTest. If you just want a Boolean test, this is slower than IString::indexOf (p. 481).

```
unsigned occurrencesOf( const char* pSearchString,
    unsigned searchLen, unsigned startPos) const;
```

Implementation

Use these members to implement this class; specifically, they initialize or set the underlying IBuffer data.

initBuffer Resets the contents from a specified buffer or buffers.

```
IString& initBuffer( double d);

IString& initBuffer( const void* p1, unsigned len1,
    const void* p2 = 0, unsigned len2 = 0,
    const void* p3 = 0, unsigned len3 = 0,
    char padChar = 0);

IString& initBuffer( long n);

IString& initBuffer( unsigned long n);
```

setBuffer Sets the private data member to point to a new IBuffer (p. 333) object.

```
IString& setBuffer( IBuffer* ibuff);
```

Queries

Use these members to access general information about the string.

buffer Returns the address of the IBuffer (p. 333) referred to by this IString.

```
IBuffer* buffer() const;
```

data Returns the address of the contents of the IString.

```
char* data() const;
```

defaultBuffer Returns a pointer to the contents of the nullBuffer data member.

```
static char* defaultBuffer();
```

lengthOf Returns the length of a C character array.

```
static unsigned lengthOf( const char* p);
```

Testing

Use these members to determine if an IString contains only characters from a predefined set.

isAbbrevFor If the receiver is a valid abbreviation of the specified string, true is returned.

The parameters are the following:

pFullString
The full string for the abbreviation check. The string can be either a NULL-terminated character string or not.

fullLen The full length of the specified *pFullString* minus the null terminator.

minLen The minimum length to match for it to be a valid abbreviation. If you specify 0, the minimum length is the length of the receiver's string.

```
Boolean isAbbrevFor( const char* pFullString,
    unsigned fullLen, unsigned minLen) const;
```

isLike If the receiver matches the specified pattern, which can contain wildcard characters, true is returned.

- You can use the first wildcard character to specify that 0 or more arbitrary characters are accepted. The default wildcard character that does this is *, but you can specify another character when calling IString::isLike. For example:

  ```
  IString( "Allison" ).isLike( "Al*ison" ) -> true
  ```

- You can use the second wildcard character to specify that a single arbitrary character is accepted. The default wildcard character that does this is ?, but you can specify another character when calling IString::isLike. For example:

  ```
  IString( "istring7.cpp" ).isLike( "i*.?pp" ) -> true
  IString( "Not a question!" ).isLike( "*?", '*', '-' ) -> false
  ```

```
Boolean isLike( const char* pPattern,
    unsigned patternLen, char zeroOrMore,
    char anyChar) const;
```

Protected Data

Utility Data

These protected static data members provide useful values for implementing IString. IString uses the various representation of null and zero for initialization and comparison purposes.

maxLong The maximum value of a long, with 32-bit unsigned long integers.

```
static const char *maxLong;
```

PM This value is "2147483647" on OS/2 with 32-bit unsigned long integers.

null A string that contains no element.

```
static const char *null;
```

nullBuffer A pointer to the null buffer's contents.

```
static char *nullBuffer;
```

zero The number 0.

```
static const char *zero;
```

Inherited Protected Data

IBase		
recoverable	**unrecoverable**	

Nested Type Definitions

BitOperator

```
typedef enum { and , or , exclusiveOr } BitOperator;
```

Use these enumerators to specify the bit operator to apply to the applyBitOp function. Valid bit operators are as follows:

- and
- or
- exclusiveOr

IndexType

```
typedef enum { charIndex , wordIndex } IndexType;
```

These enumerators specify whether the result from the findPhrase function is a word index or a character index:

charIndex
: Returns the result as the byte index within the string

wordIndex
: Returns the result as the index of the matching word. For example, the first word is 1, the second word is 2, and so forth.

Related Enumeration

BitOperator

IStringEnum

Derivation Inherits from none.

Inherited By None.

Header File istrenum.hpp

The IStringEnum class serves as a repository for enumeration types related to the IString class. The library places these enumeration types here so they can easily be shared between code that implements the classes IString (p. 469), IBuffer (p. 333), and IDBCSBuffer (p. 361).

Nested Type Definitions

CharType

```
typedef enum { sbcs , dbcs1 = 1 , mbcs1 = 1 , dbcs2 = 2 ,
               mbcs2 = 2 , mbcs3 = 3 , mbcs4 = 4 } CharType;
```

These enumerators specify the various types of characters that comprise an IString:

sbcs
The IString contains single-byte character set (SBCS) characters.

dbcs1
The IString contains the first byte of a double-byte character support (DBCS) character.

dbcs2
The IString contains the second byte of a double-byte character support (DBCS) character.

StripMode

```
typedef enum { leading , trailing , both } StripMode;
```

Enumeration that defines the mode of various functions that strip leading characters, trailing characters, or both from IStrings.

Related Enumeration

CharType

IStringParser

Derivation

IBase
  IStringParser

Inherited By None.

Header File istparse.hpp

Members

Member	Page	Member	Page
Constructor	512	operator >>	505
operator <<	504	~IStringParser	506

Objects of this class enable you to parse the content of an IString (p. 469) and place portions of the string into other strings. You can limit the parsing of a string by specifying the following:

- Patterns that must be matched
- Relative or absolute column numbers

This class's functions work much like the REXX parse statement.

Typically, you create IStringParser objects implicitly by applying the right-shift operator to an IString. IStringParser also provides the right-shift operator as a member function so you can chain together invocations of the operator. For example, a typical expression using IStringParser objects might look like the following:

```
aFileName >> drive >> ':' >> path;
```

The right-shift operator does one of four things, depending on the type of the right-hand operand:

IString The string parser object sets this string to the next token from the text being parsed.

pattern The parser advances to the next character beyond the occurrence of that pattern in its text. The pattern can be any of the following:

const char*
Searches for the sequence of characters described by the character array.

const IString
Searches for the sequence of characters described by the string. Note that the treatment of a const IString is fundamentally different from the treatment of a non-const IString.

char Searches for the next occurrence of the specified character.

IStringTest
Searches for the next character in the text for which the string test object returns true.

number The current parser text position is adjusted by the specified amount. The value can be positive or negative.

special IStringParser defines special right-shift operands that perform the following special-purpose parser operations:

IStringParser::reset This enumerator resets the parser text position to 1.

IStringParser::skip This enumerator skips one token in the text. It is equivalent to *>> temp*, where temp is a temporary IString that is discarded. This is equivalent to using '.' in REXX.

IStringParser::Skip An object of this class skips a given number of tokens.

You can also use the left-shift operator with an unsigned numeric parameter. This repositions the parser object to the specified column. Note that the parameter is not relative as it is in the case of the right-shift operator. Instead, it is an absolute column position.

Public Functions

Absolute Column Positioning

Use these members to reset the parser text position to an absolute column number.

operator << Changes the parser text position to an absolute column number. This is a left-shift operator.

```
IStringParser& operator <<( unsigned long position);
```

Commands

Use these members to permit special-purpose parsing techniques. They allow you to handle special commands and to skip objects.

operator >> Parses the text string. The right-shift operator is the primary function for parsing the text string. The library overloads this function so you can specify how you want the text string parsed via the type of parameter accepted by a particular overload.

1 `IStringParser& operator >>( const SkipWords& skipObject);`

Skips the next *n* words in the parser text, where *n* is the number of words specified when constructing the IStringParser::SkipWords (p. 513) object.

2 `IStringParser& operator >>( Command command);`

Resets the parser text position as follows:

- To the beginning of the text
- To skip the next token in the parser text

Use the enumeration IStringParser::Command (p. 512) to specify the parsing token.

3 `IStringParser& operator >>( IString& token);`

Parses the next token from the object into the IString object. This parameter places the rest of the parser text into the IString object. When the parser encounters a subsequent parsing instruction, it adjusts the token placed into the string. For example:

```
token1 token2 >> token1   // token1 == "token1 token2" at this point
              >> token2;       // token2 == token2 and
                                 // token1  == token1.
```

4 `IStringParser& operator >>( const IString& pattern);`

Finds a matching pattern within the parser text and moves the parser text position. If the pattern is not found, the parser moves the position off the end of the parser text.

5 `IStringParser& operator >>( const char* pattern);`

Finds a matching pattern within the parser text and moves the parser text position. If the pattern is not found, the parser moves the position off the end of the parser text.

6 `IStringParser& operator >>( char pattern);`

Finds a matching pattern within the parser text and moves the parser text position. If the pattern is not found, the parser moves the position off the end of the parser text.

7 `IStringParser& operator >>( const IStringTest& test);`

Applies the IStringTest object to the parser text and moves the parser text position to the next character that satisfies the string test. If the string test is not satisfied, the parser moves the position off the end of the parser text.

8 `IStringParser& operator >>( int delta);`

Moves the parser text position relative to the current parser text position. For example:

```
"1234" >> token1 >> 1 >> token2 >> 2 >> token3;
```

results in:

```
token1 == "1"
token2 == "23"
token3 == "4"
```

9 `IStringParser& operator >>( unsigned long delta);`

Moves the parser text position relative to the current parser text position. For example:

```
"1234" >> token1 >> 1 >> token2 >> 2 >> token3;
```

results in:

```
token1 == "1"
token2 == "23"
token3 == "4"
```

Constructor and Destructor

The destructor member is the default. The constructor members are protected to prevent you from creating objects except via use of the shift operators.

You can construct a string parser object by providing:

- a string that defines the text to be parsed
- an existing parser object (copy constructor)

Note that usually you will construct parser objects by applying the right-shift operator to a string. The constructor is protected to prevent you from creating objects except via use of those operators. Creation is prevented because of the nature of string parser objects. Since they hold references to operands, it is unwise to permit the objects to persist beyond the scope of those operands.

Destructor `~IStringParser();`

Destructor, decrements reference count.

Pattern Matching

Use these members to advance to the next occurrence of the argument pattern in the parser text. Upon return, the parser is positioned at the next character beyond the text that matched the pattern. If the pattern is not found, the parser is positioned off the end of the text. Note that when using an IString as a pattern, you should cast it to a const IString reference.

operator >> Parses the text string. The right-shift operator is the primary function for parsing the text string. The library overloads this function so you can specify how you want the text string parsed via the type of parameter accepted by a particular overload.

1 `IStringParser& operator >>( const IStringTest& test);`

Applies the IStringTest object to the parser text and moves the parser text position to the next character that satisfies the string test. If the string test is not satisfied, the parser moves the position off the end of the parser text.

2 `IStringParser& operator >>( Command command);`

Resets the parser text position as follows:

- To the beginning of the text
- To skip the next token in the parser text

Use the enumeration IStringParser::Command (p. 512) to specify the parsing token.

3 `IStringParser& operator >>( const SkipWords& skipObject);`

Skips the next *n* words in the parser text, where *n* is the number of words specified when constructing the IStringParser::SkipWords (p. 513) object.

4 `IStringParser& operator >>( IString& token);`

Parses the next token from the object into the IString object. This parameter places the rest of the parser text into the IString object. When the parser encounters a subsequent parsing instruction, it adjusts the token placed into the string. For example:

```
token1 token2 >> token1   // token1 == "token1 token2" at this point
                >> token2;       // token2 == token2 and
                                  // token1  == token1.
```

5 `IStringParser& operator >>( const IString& pattern);`

Finds a matching pattern within the parser text and moves the parser text position. If the pattern is not found, the parser moves the position off the end of the parser text.

6 `IStringParser& operator >>( const char* pattern);`

Finds a matching pattern within the parser text and moves the parser text position. If the pattern is not found, the parser moves the position off the end of the parser text.

7 `IStringParser& operator >>( char pattern);`

Finds a matching pattern within the parser text and moves the parser text position. If the pattern is not found, the parser moves the position off the end of the parser text.

8 `IStringParser& operator >>( int delta);`

Moves the parser text position relative to the current parser text position. For example:

```
"1234" >> token1 >> 1 >> token2 >> 2 >> token3;
```

results in:

```
token1 == "1"
token2 == "23"
token3 == "4"
```

9 `IStringParser& operator >>( unsigned long delta);`

Moves the parser text position relative to the current parser text position. For example:

```
"1234" >> token1 >> 1 >> token2 >> 2 >> token3;
```

results in:

```
token1 == "1"
token2 == "23"
token3 == "4"
```

Relative Column Positioning

Use these members to move the parser text position relative to its current position. A negative argument moves backward; a positive argument moves forward. The adjustment is made starting at the point at which the prior parsing instruction started.

For example:

"1234" >> token1 >> 1 >> token2 >> 2 >> token3;

will result in:

token1 == "1"
token2 == "23"
token3 == "4".

operator >> Parses the text string. The right-shift operator is the primary function for parsing the text string. The library overloads this function so you can specify how you want the text string parsed via the type of parameter accepted by a particular overload.

1 `IStringParser& operator >>( unsigned long delta);`

Moves the parser text position relative to the current parser text position. For example:

```
"1234" >> token1 >> 1 >> token2 >> 2 >> token3;
```

results in:

```
token1 == "1"
token2 == "23"
token3 == "4"
```

2 `IStringParser& operator >>( Command command);`

Resets the parser text position as follows:

- To the beginning of the text
- To skip the next token in the parser text

Use the enumeration IStringParser::Command (p. 512) to specify the parsing token.

3 `IStringParser& operator >>( const SkipWords& skipObject);`

Skips the next *n* words in the parser text, where *n* is the number of words specified when constructing the IStringParser::SkipWords (p. 513) object.

4 `IStringParser& operator >>( IString& token);`

Parses the next token from the object into the IString object. This parameter places the rest of the parser text into the IString object. When the parser encounters a subsequent parsing instruction, it adjusts the token placed into the string. For example:

```
token1 token2 >> token1   // token1 == "token1 token2" at this point
              >> token2;      // token2 == token2 and
                                  // token1  == token1.
```

5 `IStringParser& operator >>( const IString& pattern);`

Finds a matching pattern within the parser text and moves the parser text position. If the pattern is not found, the parser moves the position off the end of the parser text.

6 `IStringParser& operator >>( const char* pattern);`

Finds a matching pattern within the parser text and moves the parser text position. If the pattern is not found, the parser moves the position off the end of the parser text.

7 `IStringParser& operator >>( char pattern);`

Finds a matching pattern within the parser text and moves the parser text position. If the pattern is not found, the parser moves the position off the end of the parser text.

8 `IStringParser& operator >>( const IStringTest& test);`

Applies the IStringTest object to the parser text and moves the parser text position to the next character that satisfies the string test. If the string test is not satisfied, the parser moves the position off the end of the parser text.

9 `IStringParser& operator >>( int delta);`

Moves the parser text position relative to the current parser text position. For example:

```
"1234" >> token1 >> 1 >> token2 >> 2 >> token3;
```

results in:

```
token1 == "1"
token2 == "23"
token3 == "4"
```

Tokens

Use these members to parse the next token from the parser object and place it into the IString operand. By necessity, these members place the rest of the parser text into the string. When the parser encounters a subsequent parsing instruction, it goes back and adjusts the token placed into the string.

For example:

```
"token1 token2" >> token1    // token1 == token1 token2 at this point
                >> token2;          // token2 == "token2" and
                                        // token1 == "token1".
```

operator >> Parses the text string. The right-shift operator is the primary function for parsing the text string. The library overloads this function so you can specify how you want the text string parsed via the type of parameter accepted by a particular overload.

1 `IStringParser& operator >>( IString& token);`

Parses the next token from the object into the IString object. This parameter places the rest of the parser text into the IString object. When the parser encounters a subsequent parsing instruction, it adjusts the token placed into the string. For example:

```
token1 token2 >> token1   // token1 == "token1 token2" at this point
                >> token2;       // token2 == token2 and
                                    // token1  == token1.
```

2 `IStringParser& operator >>( Command command);`

Resets the parser text position as follows:

- To the beginning of the text
- To skip the next token in the parser text

Use the enumeration IStringParser::Command (p. 512) to specify the parsing token.

3 `IStringParser& operator >>( const SkipWords& skipObject);`

Skips the next *n* words in the parser text, where *n* is the number of words specified when constructing the IStringParser::SkipWords (p. 513) object.

4 `IStringParser& operator >>( const IString& pattern);`

Finds a matching pattern within the parser text and moves the parser text position. If the pattern is not found, the parser moves the position off the end of the parser text.

5 `IStringParser& operator >>( const char* pattern);`

Finds a matching pattern within the parser text and moves the parser text position. If the pattern is not found, the parser moves the position off the end of the parser text.

6 `IStringParser& operator >>( char pattern);`

Finds a matching pattern within the parser text and moves the parser text position. If the pattern is not found, the parser moves the position off the end of the parser text.

7 `IStringParser& operator >>( const IStringTest& test);`

Applies the IStringTest object to the parser text and moves the parser text position to the next character that satisfies the string test. If the string test is not satisfied, the parser moves the position off the end of the parser text.

8 `IStringParser& operator >>( int delta);`

Moves the parser text position relative to the current parser text position. For example:

```
"1234" >> token1 >> 1 >> token2 >> 2 >> token3;
```

results in:

```
token1 == "1"
token2 == "23"
token3 == "4"
```

9 `IStringParser& operator >>( unsigned long delta);`

Moves the parser text position relative to the current parser text position. For example:

```
"1234" >> token1 >> 1 >> token2 >> 2 >> token3;
```

results in:

```
token1 == "1"
token2 == "23"
token3 == "4"
```

Inherited Public Functions

IBase		
asDebugInfo	**messageFile**	**setMessageFile**
asString	**messageText**	**version**

Protected Functions

Constructors

The destructor member is the default. The constructor members are protected to prevent you from creating objects except via use of the shift operators.

You can construct a string parser object by providing:

- a string that defines the text to be parsed
- an existing parser object (copy constructor)

Note that usually you will construct parser objects by applying the right-shift operator to a string. The constructor is protected to prevent you from creating objects except via use of those operators. Creation is prevented because of the nature of string parser objects. Since they hold references to operands, it is unwise to permit the objects to persist beyond the scope of those operands.

Constructors

1
```
IStringParser( const IStringParser& parser);
```

Construct an object from an existing IStringParser object. The IStringParser object specifies the text string to parse. This constructor increments the usage count of the IStringParser object.

2
```
IStringParser( const IString& text);
```

Construct an object from an IString object. The IString object specifies the text string to parse.

Inherited Protected Data

IBase		
recoverable	**unrecoverable**	

Nested Classes

IStringParser contains the following nested classes:

IStringParser::SkipWords (see page 513)

Command

```
Command { reset, skipWord, skip = skipWord };
```

These enumerators specify special purpose parsing tokens:

reset	Resets the parser position to 1.
skip	Causes the parser to skip one token (that is, a word) in the input text.

IStringParser::SkipWords

Derivation

IBase
  IStringParser::SkipWords

Inherited By None.

Header File istparse.hpp

Members

Member	Page
Constructor	513
numberOfWords	513
SkipWords	513

Objects of the nested class IStringParser::SkipWords skip a specified number of words in the input text without assigning those words to output strings. Use these objects when parsing text with the class IStringParser (p. 503).

Public Functions

Constructors

You can construct objects of this class by specifying the number of words to skip. Use in conjunction with IStringParser objects to parse the content of an IString (p. 469) and place portions of the string into other strings.

Constructor You can construct objects of this class by specifying the number of words to skip. The default is one word.

```
SkipWords( unsigned long numberOfWords = 1);
```

Word Functions

Use these members to retrieve the number of words to skip. You set the number of words to skip in the constructor.

numberOfWords

Returns the number of words to skip.

```
unsigned long numberOfWords() const;
```

Inherited Public Functions

IBase		
asDebugInfo	**messageFile**	**setMessageFile**
asString	**messageText**	**version**

Inherited Protected Data

IBase		
recoverable	**unrecoverable**	

IStringTest

Derivation

IBase
  IVBase
    IStringTest

Inherited By IStringTestMemberFn

Header File istrtest.hpp

Members

Member	Page	Member	Page
Constructor	516	type	517
data	517	˜IStringTest	516
test	516		

The IStringTest class defines the basic protocol for test objects that you can pass to IStrings (p. 469) or I0Strings (p. 307) to assist in performing various test and search functions. This class also provides concrete implementation for the common case of using a C function for such testing.

The library provides a derived template class, IStringTestMemberFn (p. 519), to facilitate using member functions of any class on the IString functions that support IStringTest.

Derived classes should re-implement the virtual function IStringTest::test (p. 516) to test characters passed by the IString and return the appropriate result.

A constructor for this class accepts a pointer to a C function that in turn accepts an integer as a parameter and returns a Boolean. You can use such functions anywhere an IStringTest can be used. Note that this is the type of the standard C library "is" functions that check the type of C characters.

Public Functions

Constructor and Destructor

You can construct and destruct objects of this class with a pointer to the C function to be used to implement the member IStringTest::test (p. 516). Such members can be used anywhere an

IStringTest can be used. Note that these members are the same as the standard C library is.... functions that check the type of C characters.

This class also provides a protected constructor, which derived classes can use to reuse the space for the C/C++ function pointer.

Constructors

1 `IStringTest( CFunction& cFunc);`

Accepts a pointer to a C function.

2 `IStringTest( CPPFunction& cppFunc);`

Accepts a pointer to a C++ function.

Destructor `~IStringTest();`

Testing

Use these members to implement an actual test.

test Tests the specified integer (character) and returns true or false as returned by the C function provided at construction. Derived classes should override this function to implement their own testing function.

```
virtual Boolean test( int c) const;
```

Inherited Public Functions

IVBase		
asDebugInfo	asString	

IBase		
asDebugInfo	**messageFile**	**setMessageFile**
asString	**messageText**	**version**

Protected Functions

Constructor

You can construct and destruct objects of this class with a pointer to the C function to be used to implement the member IStringTest::test (p. 516). Such members can be used anywhere an

IStringTest can be used. Note that these members are the same as the standard C library is.... functions that check the type of C characters.

This class also provides a protected constructor, which derived classes can use to reuse the space for the C/C++ function pointer.

IStringTest

```
IStringTest( FnType type, void* userData);
```

Used by derived classes to reuse the space for the C/C++ function pointer.

Protected Data

Test Function Description

Use these members to implement this class.

data

Data member union, varying by FnType:
cFn - Pointer to a C function.
user - Pointer to an arbitrary derived-class data (if FnType is neither c nor cpp).

```
union { CFunction *cFn; CPPFunction *cppFn; void *user; } data;
```

type

Data member FnType. FnType is an enumeration describing the various flavors of functions supported; user-defined, C, C++ static or non-member function, C++ member function, const C++ member function.

```
FnType type;
```

Inherited Protected Data

IBase		
recoverable	**unrecoverable**	

Nested Type Definitions

FnType

```
FnType { user, c, cpp, memFn, cMemFn };
```

Use these enumerators to specify the type of functions supported:

user	User-defined.
c	C.

cpp	C++ static or non-member function.
memFn	C++ member function.
cMemFn	Const C++ member function.

CFunction

```
typedef ICStrTestFn CFunction;
```

Pointer to the C function that accepts an integer parameter and returns Boolean.

(int)

```
typedef Boolean CPPFunction ( int );
```

Pointer to plain (static or non-member) C++ function accepting integer argument and returning Boolean.

IStringTestMemberFn

Derivation

IBase
  IVBase
    IStringTest
      IStringTestMemberFn

Inherited By None.

Header File istrtest.hpp

Members

Member	Page
Constructor	520
test	520

The library provides the template class IStringTestMemberFn as an IStringTest-type wrapper for particular C++ member functions. Doing so lets you use such member functions in conjunction with functions from IString (p. 469) and I0String (p. 307) that accept an IStringTest (p. 515) object as an parameter.

Customization (Template Argument)

IStringTestMemberFn is a template class that is instantiated with the following template argument:

T The class of object whose member function is to be wrappered.

Public Functions

Constructors

You can construct objects of this class in the following ways:

- Use the constructor that supports const member functions.
- Use the constructor that supports nonconst member functions. You must specify a nonconst member function as the first parameter.

Both constructors for the object require the following:

- An object of the class T (nonconst object for nonconst member functions).

- A pointer to a member function of the class T. The library applies this member function to the specified object to test each character passed to the test member of this class. The member function must accept a single integer parameter and return a Boolean.

Constructors

1 `IStringTestMemberFn( T& object, NonconstFn nonconstFn);`

Use this for the non-const member functions. The object of the class T must be non-const.

2 `IStringTestMemberFn( const T& object, ConstFn constFn);`

Use this for the const member functions.

Testing

Use these members to dispatch member functions.

test Overridden to dispatch a member function against an object.

```
virtual Boolean test( int c) const;
```

Inherited Public Functions

IStringTest		
test		

IVBase		
asDebugInfo	asString	

IBase		
asDebugInfo	**messageFile**	**setMessageFile**
asString	**messageText**	**version**

Inherited Protected Data

IStringTest		
data	type	

IBase		
recoverable	**unrecoverable**	

Nested Type Definitions

(int)

```
typedef Boolean ( T::* NonconstFn ) ( int );
```

Non-const member function of the appropriate type.

const

```
typedef Boolean ( T::* ConstFn ) ( int ) const;
```

const member function of the appropriate type.

ISystemErrorInfo

Derivation

```
IBase
  IVBase
    IErrorInfo
      ISystemErrorInfo
```

Inherited By

None.

Header File

iexcept.hpp

Members

Member	Page	Member	Page
Constructor	524	text	525
errorId	524	throwSystemError	525
isAvailable	524	˜ISystemErrorInfo	524
operator const char *	524		

Objects of the ISystemErrorInfo class represent error information that you can include in an exception object. When an OS/2 DOS system call results in an error condition, objects of the ISystemErrorInfo class are created. You can use the error text to construct a derived class object of IException (p. 379).

The library provides the ITHROWSYSTEMERROR macro for throwing exceptions constructed with the following ISystemErrorInfo information:

- 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 enumeration IErrorInfo::ExceptionType (p. 377), which specifies the type of exception this macro creates
- One of the values of the enumeration IException::Severity (p. 386), which specifies the severity of the exception

This macro generates code that calls throwSystemError (p. 525), which does the following:

1. Creates an ISystemErrorInfo object
2. Uses the object to create an IException object
3. Adds the operatingSystem error group to the object
4. Adds location information
5. Logs the exception data

6. Throws the exception

Motif You can create objects of this class on AIX, but the objects contain no useful information and only have the default message: "System exception condition detected."

Public Functions

Constructor and Destructor

You can construct and destruct objects of this class. You cannot copy or assign objects of this class.

Constructor
```
ISystemErrorInfo( unsigned long systemErrorId,
    const char* systemFunctionName = 0);
```

You can only construct objects of this class using the default constructor.

Note: If the constructor cannot load the error text, the library provides the following default text: “No error text is available.”.

systemErrorId
The error ID identifying an operating system error.

systemFunctionName
(Optional) The name of the failing system call that returned the error ID. If you specify *systemFunctionName*, the constructor prefixes it to the error text.

Destructor
```
virtual ~ISystemErrorInfo();
```

Error Information

Use these members to return error information provided by objects of this class.

errorId Returns the error ID.
```
virtual unsigned long errorId() const;
```

isAvailable If the error information is available, true is returned.
```
virtual Boolean isAvailable() const;
```

operator const char *
Returns the error text.
```
virtual operator const char *() const;
```

text
Returns the error text.

```
virtual const char* text() const;
```

Throw Support

Use these members to support the throwing of exceptions.

throwSystemError

This function is used by the ITHROWSYSTEMERROR macro. The function creates an ISystemErrorInfo object and uses the text from it to do the following:

1. Create an exception object
2. Add the location information to it
3. Log the exception data
4. Throw the exception

systemErrorId
The error ID from the system.

functionName
The name of the function where the exception occurred.

location
An IExceptionLocation (p. 389) object containing the following:

- Function name
- File name
- Line number where the function is called

name
Use the enumeration IErrorInfo::ExceptionType (p. 377) to specify the type of the exception. The default is accessError.

severity
Use the enumeration IException::Severity (p. 386) to specify the severity of the error. The default is recoverable.

```
static void throwSystemError( unsigned long systemErrorId,
    const char* functionName,
    const IExceptionLocation& location,
    IErrorInfo::ExceptionType name = accessError,
    IException::Severity severity = recoverable);
```

Inherited Public Functions

IErrorInfo		
errorId	isAvailable	operator const char *

ISystemErrorInfo

IVBase		
asDebugInfo	asString	

IBase		
asDebugInfo	**messageFile**	**setMessageFile**
asString	**messageText**	**version**

Inherited Protected Data

IBase		
recoverable	**unrecoverable**	

ITime

Derivation

IBase
  ITime

Inherited By None.

Header File itime.hpp

Members

Member	Page	Member	Page
Constructor	529	operator +=	530
asCTIME	530	operator -	530
asSeconds	530	operator -=	530
asString	529	operator <	528
hours	531	operator <=	528
initialize	531	operator ==	528
minutes	531	operator >	528
now	529	operator >=	528
operator !=	528	seconds	531
operator +	530		

Objects of the ITime class represent units of time (hours, minutes, and seconds) as portions of days and provide support for converting these units of time into numeric and ASCII format. You can compare and operate on ITime objects by adding them to and subtracting them from other ITime objects.

A related class whose objects also represent units of time is the class IDate (p. 351).

The ITime class returns locale-sensitive information, based on the current locale defined at runtime. See the description of the standard C function setlocale in the OS/2 system documentation for information about setting the locale.

Public Functions

Comparisons

Use these members to compare two ITime objects. Use any of the full complement of comparison operators and applying the natural meaning.

operator != Compares two objects to determine whether they are not equal.

```
Boolean operator !=( const ITime& aTime) const;
```

operator < Compares two objects to determine whether one is less than the other.

```
Boolean operator <( const ITime& aTime) const;
```

operator <= Compares two objects to determine whether one is less than or equal to the other.

```
Boolean operator <=( const ITime& aTime) const;
```

operator == Compares two objects to determine whether they are equal.

```
Boolean operator ==( const ITime& aTime) const;
```

operator > Compares two objects to determine whether one is greater than the other.

```
Boolean operator >( const ITime& aTime) const;
```

operator >= Compares two objects to determine whether one is greater than or equal to the other.

```
Boolean operator >=( const ITime& aTime) const;
```

Constructors

You can construct objects of this class in the following ways:

- Use the default constructor, which returns the current time.
- Give the number of seconds since midnight that the time represents. In this case, the number of seconds can be negative and is subtracted from the number of seconds in a day.
- Give the number of hours, minutes, and seconds since midnight that the time represents. In this case, the number of seconds cannot be negative.
- Copy another ITime object.
- Give a container details CTIME structure.

Constructors

1 `ITime( const ITime& aTime);`

Use this constructor to copy another ITime object.

2 `ITime();`

Using this constructor returns the current time; it's the default.

3 `ITime( long seconds);`

Use this constructor by specifying the number of seconds since midnight that the time is to represent. For negative values, the constructor subtracts that value from the number of seconds in a day.

4 `ITime( unsigned hours, unsigned minutes, unsigned seconds = 0);`

Specify the number of hours, minutes, and seconds since midnight that the time represents. The number of seconds cannot be negative.

5 `ITime( const _CTIME& cTime);`

Supported On:
PM

You use this constructor to construct an ITime object from a container details CTIME structure.

Current Time

Use this member when you need the current time.

now

Returns the current time.

Note: You can use this function as an ITime constructor.

```
static ITime now();
```

Diagnostics

Use these members to provide an IString representation for an ITime object and the capability to output the object to a stream. The formatting is based on the strftime conversion specifications. Often, you use these members to write trace information when debugging your code.

asString

Returns the ITime object as a string that is formatted according to the specified format. This format string can contain time "conversion specifiers" as defined for the standard C library function strftime in the TIME.H header file. The default format is %X, which yields the time as hh:mm:ss. Refer to the *VisualAge C++: C Library Reference* for more information about the strftime function.

The conversion specifiers that apply to ITime and their meanings are listed in the following table. IDate::asString (p. 354) describes conversion specifiers that apply to dates.

Specifier	Meaning
%c	Insert date and time of locale.
%H	Insert hour (24-hour clock) as a decimal number (00-23).
%I	Insert hour (12-hour clock) as a decimal number (01-12).
%M	Insert minute (00-59).
%p	Insert equivalent of either AM or PM locale.
%S	Insert second (00-61).
%X	Insert time representation of locale.
%Z	Insert name of time zone, or no characters if time zone is not available.
%%	Insert %.

```
IString asString( const char* fmt = " % X") const;
```

Manipulation

Use these members to update an ITime object by adding or subtracting another ITime object. Use any of the full complement of addition or subtraction operators and apply the natural meaning.

operator + Adds two objects.

```
ITime operator +( const ITime& aTime) const;
```

operator += Adds two objects and stores the result in the receiver.

```
ITime& operator +=( const ITime& aTime);
```

operator - Subtracts one object from another.

```
ITime operator -( const ITime& aTime) const;
```

operator -= Subtracts one object from another and stores the result in the receiver.

```
ITime& operator -=( const ITime& aTime);
```

Time Queries

Use these members to access the seconds, minutes and hours of an ITime object.

asCTIME Returns the time as a container CTIME structure.

```
_CTIME asCTIME() const;
```

Supported On:
PM

asSeconds Returns the number of seconds since midnight.

```
long asSeconds() const;
```

hours Returns the number of hours past midnight.

```
unsigned hours() const;
```

minutes Returns the number of minutes past the hour.

```
unsigned minutes() const;
```

seconds Returns the number of seconds past the minute.

```
unsigned seconds() const;
```

Inherited Public Functions

IBase		
asDebugInfo	**messageFile**	**setMessageFile**
asString	**messageText**	**version**

Protected Functions

Implementation

Use these members to initialize objects of this class.

initialize A common initialization function used by the ITime constructors.

```
ITime& initialize( long seconds);
```

Inherited Protected Data

IBase		
recoverable	**unrecoverable**	

ITrace

Derivation

IBase
  IVBase
    ITrace

Inherited By None.

Header File itrace.hpp

Members

Member	Page	Member	Page
Constructor	535	threadId	538
disableTrace	536	traceDestination	537
disableWriteLineNumber	536	write	537
disableWritePrefix	536	writeFormattedString	538
enableTrace	536	writeString	538
enableWriteLineNumber	536	writeToQueue	537
enableWritePrefix	536	writeToStandardError	537
isTraceEnabled	536	writeToStandardOutput	537
isWriteLineNumberEnabled	536	˜ITrace	535
isWritePrefixEnabled	536		

Objects of the ITrace class provide module tracing within the library. Whenever an exception is thrown by the library, trace records are output with information about the exception. You can use the *ICLUI_TRACE* and *ICLUI_TRACETO* environment variables to redirect the trace output to a file. The output trace records contain the following:

- Error message text
- Error ID
- Class name
- Information from the class IExceptionLocation (p. 389)

The Data Type and Exception Class Library throws only two exceptions:

ID	**Explanation**
1010	`IC_ISTRING_OVERFLOW`
1011	`IC_ISTRING_INDEX_ERROR`

These error numbers are defined in the header file `icconst.h`.

For exceptions thrown by the User Interface Class Library, the value of the error ID is one of the following:

- The value of `WinGetLastError` or `ERRINFO.idError` if the error is an OS/2 PM-related error.
- A hardcoded 0, if the exception is an X/Motif-related error. In most cases, these window management systems do not give any error ID for the exception to pass on.
- The throwing function, which typically throws the exception after performing a system call, if the exception is a system error.

For exceptions thrown by the Collection Class Library, the error ID contains the letters `CCL`, then four numeric digits, then the letter `E`.

Also by default, the library disables tracing. You can set tracing on by entering *ICLUI_TRACE=ON* in the environment.

By default, the library attaches a prefix to the trace entry containing a sequence number followed by the process and thread where the trace call occurred. You can remove prefix area tracing by entering *ICLUI_TRACE=NOPREFIX* in the environment. Doing so has the side effect of turning tracing on.

You can set the output location of tracing by entering one of the following in the environment:

- *ICLUI_TRACETO=STDERR* for the standard error stream (stderr)
- *ICLUI_TRACETO=STDOUT* for the standard output (stdout)
- *ICLUI_TRACETO=QUEUE* for a queue

Specifying any of the preceding locations has the side effect of turning tracing on.

In addition to turning the trace options on and off in the environment, the library also provides static member functions to do the same thing under program control.

The library supports trace input as IStrings or character arrays, and the library automatically adds a line feed on all trace calls.

To enable you to compile the trace calls in and out of your code, the library provides the following sets of macros for tracing modules and data:

- The library defines IC_TRACE_RUNTIME by default. The following macros are expanded:

  ```
  IMODTRACE_RUNTIME()  IFUNCTRACE_RUNTIME()  ITRACE_RUNTIME()
  ```

- If you define IC_TRACE_DEVELOP, the following macros, in addition to the RUNTIME macros, are expanded:

```
IMODTRACE_DEVELOP()  IFUNCTRACE_DEVELOP()  ITRACE_DEVELOP()
```

- If you define IC_TRACE_ALL, the following macros, in addition to the RUNTIME and DEVELOP macros, are expanded:

```
IMODTRACE_ALL()      IFUNCTRACE_ALL()      ITRACE_ALL()
```

The IMODTRACE version of the macros accepts as input a module name that it uses for construction and destruction tracing.

The IFUNCTRACE version of the macros accepts no input and uses the predefined identifier __FUNCTION__ for construction and destruction tracing.

The ITRACE version of the macros accepts a text string to be written out.

PM In OS/2, the library supports the environment variables ICLUI TRACE and ICLUI TRACETO, in addition to ICLUI_TRACE and ICLUI_TRACETO.

The default output location of tracing is the OS/2 queue \\QUEUES\\PRINTF32. You can display this queue using the program PMPRTF32.EXE.

Motif The default output location of tracing is standardOutput. Setting the output location of tracing to queue has the same effect in X/Motif as setting it to standardOutput.

Public Functions

Constructor and Destructor

You can construct objects of this class by using the default constructor. If you do not specify the optional values, this constructor creates an ITrace object, but no logging occurs on construction or destruction.

Constructor

```
ITrace( const char* traceName = 0, long lineNumber = 0);
```

You pass the optional parameters to gain the following trace behavior:

traceName (Optional) If you specify *traceName*, the name is written on construction and again on destruction.

Warning: If you pass an IString (p. 469) to the trace object, you must ensure that the lifetime of the IString exceeds the lifetime of the ITrace object. The library does not support the use of temporary IStrings.

lineNumber
(Optional) The line number where the trace statement occurred.

Destructor

```
~ITrace();
```

Enabling and Disabling

Use these members to enable or disable tracing, as well as to query whether tracing is on.

disableTrace Disables trace entries from being written.

```
static void disableTrace();
```

enableTrace Enables trace entries to be written.

```
static void enableTrace();
```

isTraceEnabled

Determines whether tracing is currently enabled.

```
static Boolean isTraceEnabled();
```

Format

Use these members to enable, disable, and query the formatting options for writing trace output.

disableWriteLineNumber

Disables the tracing of line number information.

```
static void disableWriteLineNumber();
```

disableWritePrefix

Disables the writing of the process ID, the thread ID, and the output line number to trace.

```
static void disableWritePrefix();
```

enableWriteLineNumber

Enables the tracing of line number information.

```
static void enableWriteLineNumber();
```

enableWritePrefix

Enables the writing of the process ID, the thread ID, and the output line number to trace.

```
static void enableWritePrefix();
```

isWriteLineNumberEnabled

Determines whether line numbers are currently being written.

```
static Boolean isWriteLineNumberEnabled();
```

isWritePrefixEnabled

Determines whether the line count prefix is being written.

```
static Boolean isWritePrefixEnabled();
```

Output Operations

Use these members to do the following:

- Write trace data to the current trace location
- Query the current trace location
- Set the current trace location

traceDestination

Returns the trace output destination for this trace object. The returned value is an enumerator provided by ITrace::Destination (p. 539).

```
static ITrace::Destination traceDestination();
```

write

Writes the specified text.

text	The text to write as a character string.
text	The text to write as an IString (p. 469).

```
static void write( const IString& text);
static void write( const char* text);
```

writeToQueue

Sets the location for output to \\QUEUES\\PRINTF32.

```
static void writeToQueue();
```

Motif In AIX, this member function is equivalent to writeToStandardOutput (p. 537).

writeToStandardError

Sets the location for output to the standard error stream.

```
static void writeToStandardError();
```

writeToStandardOutput

Sets the location for output to the standard output stream. Using this function is equivalent to setting the environment variable *ICLUI_TRACETO=OUT*.

Note: STDOUT is a synonym for OUT.

```
static void writeToStandardOutput();
```

Inherited Public Functions

IVBase		
asDebugInfo	asString	

IBase		
asDebugInfo	**messageFile**	**setMessageFile**
asString	**messageText**	**version**

Protected Functions

Output Operations

Use these members to do the following:

- Write trace data to the current trace location
- Query the current trace location
- Set the current trace location

writeFormattedString

Writes the trace data after formatting, which includes the following:

- Adding the prefix, if necessary
- Updating any new lines embedded in the string to include the prefix

string	Any trace information you want to write.
marker	When the library uses this function, it specifies a character to mark, or distinguish, whether the trace statement is entering (+) or exiting (-) a function. You can specify *marker* for any purpose.

```
static void writeFormattedString( const IString& string,
    char* marker);
```

writeString

Writes to the output device without formatting.

text	Any trace information you want to write.

```
static void writeString( char* text);
```

Thread ID

Use these members to query the thread ID.

threadId

Returns the current thread identifier.

```
static unsigned long threadId();
```

Motif In environments that do not support kernel threads, this function always returns a 1.

Inherited Protected Data

IBase		
recoverable	**unrecoverable**	

Destination

```
Destination { queue, standardError, standardOutput };
```

These enumerators specify the destination of the trace data:

queue	Sends the trace data to the queue.
standardError	Sends the trace data to the standard error stream (stderr).
standardOutput	Sends the trace data to the standard output (stdout).

When used on the following platforms, the queue enumerator is not supported, and queue tracing goes to stdout:

- AIX
- Solaris
- MVS

Motif AIX does not support the queue enumerator. If the trace destination is queue, tracing goes to stdout.

IVBase

Derivation

IBase
  IVBase

Inherited By

IApplication
IBaseComboBox::Cursor
IBaseListBox::Cursor
IBuffer
IClipboard
IClipboard::Cursor
IColor
IContainerColumn
IContainerControl::ColumnCursor
IContainerControl::CompareFn
IContainerControl::FilterFn
IContainerControl::Iterator
IContainerControl::ObjectCursor
IContainerControl::TextCursor
IContainerObject
IDMImage
IDMItemProvider
IDMRenderer
IErrorInfo
IEvent
IFont
IFont::FaceNameCursor
IFont::PointSizeCursor
IGList::Cursor
IGraphic
IGraphicContext
IHandler
IMenu::Cursor
IMessageBox
IMMAudioCDContents
IMMAudioCDContents::Cursor
IMMSpeed
IMMTime
INotebook::Cursor
INotebook::PageSettings
INotifier
IObserver
IObserverList
IObserverList::Cursor
IProfile
IProfile::Cursor
IRefCounted
IResource
IResourceLibrary
IResourceLock
ISpinButton::Cursor
IStringTest
ISubmenu::Cursor
ITextSpinButton::Cursor
IThread
IThread::Cursor
ITimer
ITimer::Cursor
IToolBar::FrameCursor
IToolBar::WindowCursor
ITrace
IWindow::BidiSettings
IWindow::ChildCursor
IWindow::ExceptionFn

Header File

ivbase.hpp

Members

Member	Page
asDebugInfo	542
asString	542
˜IVBase	542

The IVBase class provides basic generic behavior for all the library classes that have virtual functions. In addition, it allows derived classes to exploit the nested type and value names in the IBase class, such as Boolean, true, and false. See IBase (p. 323) for information about that class.

Derived classes are expected to override the virtual functions IVBase::asString and IVBase::asDebugInfo. This enables automatic support for the output of derived class objects on ostreams, such as cout, cerr, or both. See asString (p. 542) and asDebugInfo (p. 542) for information about those functions.

Public Functions

Constructor and Destructor

The class provides a virtual destructor to ensure that all derived classes' destructors are also virtual.

Destructor The virtual destructor ensures that all derived classes' destructors are also virtual.

```
virtual ~IVBase();
```

Conversions

Use these members to return an IVBase object in a different form.

asDebugInfo Obtains the diagnostic version of an object's contents. Generally, this is a hex string representation of a pointer to the object.

```
virtual IString asDebugInfo() const;
```

asString Obtains the standard version of an object's contents.

```
virtual IString asString() const;
```

Inherited Public Functions

IBase		
asDebugInfo	**messageFile**	**setMessageFile**
asString	**messageText**	**version**

Inherited Protected Data

IBase		
recoverable	**unrecoverable**	

IXLibErrorInfo

Derivation

IBase
  IVBase
    IErrorInfo
      IXLibErrorInfo

Inherited By None.

Header File iexcept.hpp

Members

Member	Page	Member	Page
Constructor	544	text	545
errorId	544	throwXLibError	545
isAvailable	545	˜IXLibErrorInfo	544
operator const char *	545		

Objects of the IXLibErrorInfo class represent error information that you can include in an exception object. When an X library call results in an error condition, objects of the IXLibErrorInfo class are created. IThread registers a handler through XSetErrorHandler to do the following:

- Detect the error condition
- Save the error code

You can use this error code to obtain the information about the X library error. When you have an X library function call fail, construct an object of this class to obtain the error text. You can use the error text to construct a derived class object of IException (p. 379).

The library provides the ITHROWXLIBERROR macro for throwing exceptions constructed with IXLibErrorInfo information. This macro has the following parameters:

location	The name of the X library function returning an error code.
name	Use the enumeration ExceptionType (p. 377) to specify the type of the exception. The default is accessError.

severity Use the enumeration IException::Severity (p. 386) to specify the severity of the error. The default is recoverable.

This macro generates code that calls throwXLibError (p. 545), which does the following:

1. Creates an IXLibErrorInfo object
2. Uses the object to create an IException object
3. Adds location information
4. Logs the exception data
5. Throws the exception

PM The OS/2 release of the library does not support this class.

The IXLibErrorInfo class is provided for versions of the product that run on X/Windows-based windowing systems. On OS/2, MVS and AS/400 versions of the library, this class is not supported.

Public Functions

Constructor and Destructor

You can construct and destruct objects of this class. You cannot copy or assign objects of this class.

Constructor `IXLibErrorInfo( const char* systemFunctionName = 0);` **Supported On:** Motif

You can only construct objects of this class using the default constructor.

Note: If the constructor cannot load the error text, the library provides the following default text: "No error text is available."

systemFunctionName
(Optional) The name of the failing X library function. If you specify *systemFunctionName*, the constructor prefixes it to the error text.

Destructor `virtual ~IXLibErrorInfo();` **Supported On:** Motif

Error Information

Use these members to return error information provided by objects of this class.

errorId Returns the X error code, which you can use to obtain the error text.

```
virtual unsigned long errorId() const;
```

Supported On:
Motif

isAvailable
If the error text is available, true is returned.

```
virtual Boolean isAvailable() const;
```

Supported On:
Motif

operator const char *
Returns the error text.

```
virtual operator const char *() const;
```

Supported On:
Motif

text
Returns the error text.

```
virtual const char* text() const;
```

Supported On:
Motif

Throw Support

Use these members to support the throwing of exceptions.

throwXLibError

This function is used by the ITHROWCLIBERROR macro. The function creates an IXLibErrorInfo object and uses the text from it to do the following:

- Create an exception object
- Add the location information to it
- Log the exception data
- Throw the exception

functionName
The name of the function where the exception occurred.

location
An IExceptionLocation (p. 389) object containing the following:

- Function name
- File name
- Line number where the function is called

name
Use the enumeration IErrorInfo::ExceptionType (p. 377) to specify the type of the exception. The default is accessError.

severity
Use the enumeration IException::Severity (p. 386) to specify the severity of the error. The default is recoverable.

```
static void throwXLibError( const char* functionName,
    const IExceptionLocation& location,
    IErrorInfo::ExceptionType name = accessError,
    IException::Severity severity = recoverable);
```

Supported On:
Motif

Inherited Public Functions

IErrorInfo		
errorId	isAvailable	operator const char *

IVBase		
asDebugInfo	asString	

IBase		
asDebugInfo	**messageFile**	**setMessageFile**
asString	**messageText**	**version**

Inherited Protected Data

IBase		
recoverable	**unrecoverable**	

Part 8. Database Access Class Library

Use the database access classes to connect and disconnect from your DB2/2 database and to perform transactions in the database.

Database Access Class Library

Data Access Builder C++ Classes

This chapter describes the C++ version of the database access classes. Use these classes when you generate Visual Builder parts.

IDatastore

The IDatastore provides client connection to the database, disconnect from the database, and commit and rollback of transactions.

The following attributes are used by IDatastore:

- authentication

  The authentication is the password. When logon is performed, if the userName is a null string, authentication is not used.
- datastoreName

  This is the name of the datastore to connect to.
- userName

  When attempting a connect, if userName or authentication are not specified (""), IDatastore throws the exception IDatastoreLogonFailed if there is no current user logged on. If userName and authentication are both specified, IDatastore.connect attempts a logon if the current logged on user is different than userName.
  - If logon was performed during IDatastore.connect, IDatastore.disconnect will logoff.
  - If already connected, IDatastore.connect disconnects and reconnects (including logon if necessary).

Two classes are provided to allow IDatastore to be used within Visual Builder when building your applications. They are:

- IDatastore
- IDSConnectCanvas

These parts can be found in the file **VBDAX.VBB**.

IDatastore is a nonvisual part provided in VBDAX.VBB. This part is the visual interface to the Data Access Builder class IDatastore. It is a reusable part that can be dropped into the nonvisual portion of your application.

Public Members

Use this part to connect to the datastore when the application is started. Use the settings page for this part to set the datastoreName after placing it into the composition editor edit area. Then use the ready event of the frame to call the connect() method, and the close event of the frame to call the disconnect() method.

IDSConnectCanvas is a visual part (a child of ICanvas), provided in VBDAX.VBB.

This part provides a user interface to access the functions of IDatastore. It is a reusable part that can be dropped into the main frame window of your visual applications. Use this part when the connect panel is to be the primary window of an application, or to access IDatastore from other places within the application in addition to the connect window.

Use this part by dropping onto the canvas of the primary frame window of the application. IDatastoreInterface is an exposed attribute of this part. Use the exposed events, methods and attributes to control IDatastore in addition to the controls provided on the canvas.

Derivation

IStandardNotifier
  IDatastore

Header File

IDatastore is declared in `idsmcon.hpp`.

Members

The following methods are provided:

Method	Page	Method	Page
Constructor	550	disconnect	552
Destructor	551	isConnected	552
authentication	551	rollback	552
commit	551	setAuthentication	552
connect()	551	setDatastoreName	553
connect	551	setUserName	553
datastoreName	552	userName	553

Public Members

Constructor

```
IDatastore ();

IDatastore ( const IString& datastoreName,
                          const IString& userName = "",
                          const IString& authentication = "");
```

The IDatastore constructor allocates an object. This object is used to manage a connection to the datastore.

Public Members

Destructor

```
virtual ~IDatastore ();
```

The IDatastore destructor frees space allocated by the constructor.

authentication

```
IString
   authentication() const;
IDatastore&
   setAuthentication(const IString& aAuthentication);
```

Gets the authentication.

commit

```
void commit ()
```

Commits the transactions.

Exceptions

- `IDatastoreAccessError`
- `IDatastoreConnectionNotOpen`

connect()

```
void connect ()
```

Performs the connect using the current settings specified. If there is already a connection to the database, it is disconnected and then reconnected using the current settings. For information regarding userName, see page 549.

Exceptions

- `IConnectFailed`
- `IDatastoreAccessError`
- `IDatastoreConnectionInUse`
- `IDatastoreLogoffFailed`
- `IDatastoreLogonFailed`

connect

```
void
   connect ( const IString& datastoreName,
             const IString& userName = "",
             const IString& authentication = "");
```

Performs the connect using the input parameters specified. If there is already a connection to the database, it is disconnected and then reconnected using the current settings. For information regarding userName, see page 549.

Public Members

Exceptions

- `IConnectFailed`
- `IDatastoreAccessError`
- `IDatastoreConnectionInUse`
- `IDatastoreLogoffFailed`
- `IDatastoreLogonFailed`

datastoreName

```
IString
   datastoreName() const;
```

Gets the current datastore name setting.

disconnect

```
void disconnect ()
```

Closes the connection to a database. If a logon was performed on the connect, userName is logged off.

Exceptions

- `IDatastoreConnectionNotOpen`
- `IDatastoreLogoffFailed`
- `IDisconnectError`

isConnected

```
Boolean isConnected ();
```

Returns true if there is a connection to the database.

rollback

```
void rollback ()
```

Performs a rollback on the transactions.

Exceptions

- `IDatastoreAccessError`
- `IDatastoreConnectionNotOpen`

setAuthentication

```
IDatastore&
   setAuthentication(const IString& aAuthentication);
```

Sets the authentication (password). For information regarding authentication, see page 549.

setDatastoreName

```
IDatastore&
   setDatastoreType(const IString& aDatastoreName);
```

Sets the datastore name that is used when a connection is established.

setUserName

```
IDatastore&
   setUserName(const IString& aUserName);
```

Sets the user name. For information regarding userName, see page 549.

userName

```
IString
   userName() const;
```

Gets the current user name setting. For information regarding userName, see page 549.

IPersistentObject

The IPersistentObject class provides the basic data manipulation operations where a client can call directly to add, update, delete or retrieve a row from a table. It is the abstract base class for all of the parts generated by the tool.

The generated part also contains the methods for getting and setting the attribute values. These methods are:

- <attribute type> <Attribute>() const
- const IString & <Attribute>AsString() const
- void set<attribute>(attribute type)
- void set<Attribute>(const IString &)
- Boolean is<Attribute>Nullabe() const
- Boolean is<Attribute>Null() const
- void set<Attribute>ToNull(Boolean = true)

Derivation IStandardNotifier
  IPersistentObject

Header File IPersistentObject is declared in `ipersist.hpp`.

Members The following methods are provided:

Method	Page	Method	Page
constructors	554	add	554
destructor	554	delete	554

Public Members

Method	Page	Method	Page
isDefaultReadOnly	554	operator=	555
isReadOnly	555	retrieve	555
operator!=	555	setReadOnly	555
operator==	555	update	555

Public Members

Constructors

```
IPersistentObject ();
IPersistentObject (const IPersistentObject& partCopy);
```

The IPersistentObject constructor allocates space for keeping the values of the selected columns from a single row of a table.

Destructor

```
virtual ~IPersistentObject ();
```

The IPersistentObject destructor frees the allocated space by the constructor.

add

```
virtual IPersistentObject &add () = 0;
```

Add a row to a table using the data attribute values set in the object. The object should be uniquely identified by the data identifier.

Exceptions

- `IDSAccessError`

delete

```
virtual IPersistentObject &del () = 0;
```

Delete rows from a table using the data identifier set in the object. The composition of the data identifier is defined for the concrete class.

Exceptions

- `IDSAccessError`

isDefaultReadOnly

```
virtual void isDefaultReadOnly();
```

This function returns a value of true if the table is read-only by default. Otherwise, it returns a value of false. If the table is read-only by default, an exception occurs on an add, delete or update statement. In this case, you can only use the retrieve statement to read the row from the table. Also, when the table is read-only by default, you can not use setReadOnly to change the setting.

isReadOnly

```
virtual Boolean isReadOnly();
```

This function returns a value of true if the table is read-only. Otherwise, it returns a value of false. If the table is read-only, an exception occurs on an add, delete or update statement.

operator!=

```
Boolean
 operator != (const IPersistentObject& value) const,
 operator != (const IPersistentObject* value) const;
```

Compares the values of two persistent objects.

operator==

```
Boolean
 operator == (const IPersistentObject& value) const,
 operator == (const IPersistentObject* value) const,
```

Compares the values of two persistent objects.

operator=

```
IPersistentObject&
      operator= (const IPersistentObject& aIPersistentObject);
```

This function assigns the values kept in the other object to this object.

retrieve

```
virtual IPersistentObject &retrieve () = 0;
```

Retrieve a row from a table using the data identifier set in the object. The composition of the data identifier is defined by the concrete class. The data identifier must be set on the object *before* calling retrieve.

Exceptions

- IDSAccessError

setReadOnly

```
virtual void setReadOnly (Boolean flag=true);
```

This function sets the table to read-only or updateable.

update

```
virtual IPersistentObject &update () = 0;
```

Update the row using the data identifier set in the object. The composition of the data identifier is defined by the concrete class.

Exceptions

- `IDSAccessError`

IPOManager

The IPOManager class provides operations to deal with collections of rows from a table. It is the abstract base class for all of the generated collection parts. The collection is pointed by a private data member and its data type is a pointer to IVSequence<PersistentObject*>. You can manipulate the items in the collection with the functions in the IVSequence class.

Derivation

IStandardNotifier
    IPOManager

Header File

IPOManager is declared in `ipersist.hpp`.

Members

The following methods are provided:

Method	Page	Method	Page
constructors	556	refresh	556
destructor	556	select	557
item()	556		

Public Members

Constructors

```
IPOManager();
IPOManager(const IPOManager& partCopy);
IPOManager& operator= (const IPOManager& aIPOManager);
```

The IPOManager constructor allocates space for supporting collection parts.

Destructor

```
virtual ~IPOManager();
```

The IPOManager destructor frees allocated space.

items()

```
virtual IVSequence<PersistentObject*>* items();
```

Returns the pointer to the cllection containing objects. This is a template and is not implemented in the base class.

refresh

```
virtual IPOManager &refresh () = 0;
```

Retrieves a collection of all rows on the table from the database.

Exceptions

- `IDSAccessError`

select

```
virtual IPOManager &select (const IString& clause) = 0;
```

Retrieves a collection of all rows on the table that match the specified predicate from the datastore. The predicate must be specified in the syntax of the SQL **where** clause.

Exceptions

- `IDSAccessError`

Public Members

Data Access Builder C++ Exception Classes

IConnectFailed

Derivation

```
IException
  IDatastoreAdaptorException
    IConnectFailed
```

Header File `idsexc.hpp`

Objects of the IConnectFailed class represent an exception. This class creates and throws an object when one of the following occurs:

- maximum number of IDatastoreMgr connections has already been reached
- a connection was attempted while a transaction was in progress

Constructors

```
public:
IConnectFailed(
   const char *errorText, unsigned long errorId,
   Severity severity = IException::unrecoverable);
```

You can create objects of this class by doing the following:

- Using the constructor.

errorText	The text describing this particular error.
errorId	The identifier you want to associate with this particular error.
severity	Use the enumeration IException::Severity (see page 386) to specify the severity of the error. The default is unrecoverable.

Public Members

name

```
virtual const char *name() const;
```

Returns the name of the object's class.

Inherited Public Members

For information on the members inherited from the IException class, see “IException” on page 379.

IDatastoreAccessError

Derivation

IException
    IDatastoreAccessError

Header File

`idsexc.hpp`

Objects of the IDatastoreAccessError class represent an exception. This class creates and throws an object when one of the following occurs:

- bind file not found
- connect error
- bind error.

Constructors

```
public:
IDatastoreAccessError(
   const char *errorText, unsigned long errorId,
   Severity severity = IException::unrecoverable);
```

You can create objects of this class by doing the following:

- Using the constructor.

errorText	The text describing this particular error.
errorId	The identifier you want to associate with this particular error.
severity	Use the enumeration IException::Severity (see page 386) to specify the severity of the error. The default is unrecoverable.

Public Members

name

```
virtual const char *name() const;
```

Returns the name of the object’s class.

Inherited Public Members

For information on the members inherited from the IException class, see "IException" on page 379.

IDatastoreAdaptorException

Derivation IException
  IDatastoreAdaptorException

Header File `idsexc.hpp`

Objects of the IDatastoreAdaptorException class represent an exception. This class is a generic exception message of accessing datastore.

Constructors

```
public:
IDatastoreAdaptorException(
   const char *errorText, unsigned long errorId,
   Severity severity = IException::unrecoverable);
```

You can create objects of this class by doing the following:

- Using the constructor.

errorText	The text describing this particular error.
errorId	The identifier you want to associate with this particular error.
severity	Use the enumeration IException::Severity (see page 386) to specify the severity of the error. The default is unrecoverable.

Public Members

name `virtual const char *name() const;`

Returns the name of the object's class.

Inherited Public Members

For information on the members inherited from the IException class, see "IException" on page 379.

IDatastoreConnectionInUse

Derivation IException
  IDatastoreAdaptorException
    IDatastoreConnectionInUse

Header File `idsexc.hpp`

Objects of the IDatastoreConnectionInUse class represent an exception. This class creates and throws an object when a connection is attempted using a connection which is already in use.

Constructors

```
public:
IDatastoreConnectionInUse(
   const char *errorText, unsigned long errorId,
   Severity severity = IException::unrecoverable);
```

You can create objects of this class by doing the following:

- Using the constructor.

errorText	The text describing this particular error.
errorId	The identifier you want to associate with this particular error.
severity	Use the enumeration IException::Severity (see page 386) to specify the severity of the error. The default is unrecoverable.

Public Members

name

```
virtual const char *name() const;
```

Returns the name of the object's class.

Inherited Public Members

For information on the members inherited from the IException class, see "IException" on page 379.

IDatastoreConnectionNotOpen

Derivation

IException
  IDatastoreAdaptorException
    IDatastoreConnectionNotOpen

Header File

```
idsexc.hpp
```

Objects of the IDatastoreConnectionNotOpen class represent an exception. This class creates and throws an object when an operation which requires a connection was attempted but the connection has not been established yet. For example, a call to disconnect before a connection was made.

Constructors

```
public:
IDatastoreConnectionNotOpen(
   const char *errorText, unsigned long errorId,
   Severity severity = IException::unrecoverable);
```

You can create objects of this class by doing the following:

- Using the constructor.

errorText	The text describing this particular error.
errorId	The identifier you want to associate with this particular error.
severity	Use the enumeration IException::Severity (see page 386) to specify the severity of the error. The default is unrecoverable.

Public Members

name

```
virtual const char *name() const;
```

Returns the name of the object's class.

Inherited Public Members

For information on the members inherited from the IException class, see "IException" on page 379.

IDatastoreLogoffFailed

Derivation IException
  IDatastoreLogoffFailed

Header File `idsexc.hpp`

Objects of the IDatastoreLogoffFailed class represent an exception. The class creates and throws an object when a logoff failes.

Constructors

```
public:
IDatastoreLogoffFailed(
   const char *errorText, unsigned long errorId,
   Severity severity = IException::unrecoverable);
```

You can create objects of this class by doing the following:

- Using the constructor.

errorText	The text describing this particular error.
errorId	The identifier you want to associate with this particular error.
severity	Use the enumeration IException::Severity (see page 386) to specify the severity of the error. The default is unrecoverable.

Public Members

name `virtual const char *name() const;`

Returns the name of the object's class.

Inherited Public Members

For information on the members inherited from the IException class, see "IException" on page 379.

IDatastoreLogonFailed

Derivation IException
  IDatastoreLogonFailed

Header File `idsexc.hpp`

Objects of the IDatastoreLogonFailed class represent an exception. The class creates and throws an object when a logon attempt fails.

Constructors

```
public:
IDatastoreLogonFailed(
   const char *errorText, unsigned long errorId,
   Severity severity = IException::unrecoverable);
```

You can create objects of this class by doing the following:

- Using the constructor.

errorText	The text describing this particular error.
errorId	The identifier you want to associate with this particular error.
severity	Use the enumeration IException::Severity (see page 386) to specify the severity of the error. The default is unrecoverable.

Public Members

name

```
virtual const char *name() const;
```

Returns the name of the object's class.

Inherited Public Members

For information on the members inherited from the IException class, see "IException" on page 379.

IDisconnectError

Derivation

IException
  IDatastoreAdaptorException
    IDisconnectError

Header File

```
idsexc.hpp
```

Objects of the IDisconnectError class represent an exception. This class creates and throws an object when a disconnect error occurs.

Constructors

```
public:
IDisconnectError(
   const char *errorText, unsigned long errorId,
   Severity severity = IException::unrecoverable);
```

You can create objects of this class by doing the following:

- Using the constructor.

errorText	The text describing this particular error.
errorId	The identifier you want to associate with this particular error.
severity	Use the enumeration IException::Severity (see page 386) to specify the severity of the error. The default is unrecoverable.

Public Members

name

```
virtual const char *name() const;
```

Returns the name of the object's class.

Inherited Public Members

For information on the members inherited from the IException class, see "IException" on page 379.

IDSAccessError

Derivation IException
  IDSAccessError

Header File `idsexc.hpp`

Objects of the IDSAccessError class represent an exception thrown from generated code.

When thrown, errorId is set with the following values:

```
DAX_ADD_READONLY  // Add used on a readonly table/view
DAX_ADD_SQLERR    // SQL error occurred on add
DAX_UPD_READONLY  // Update used on a readonly table/view
DAX_UPD_SQLERR    // SQL error occurred on update
DAX_DEL_READONLY  // Delete used ona readonly table/view
DAX_DEL_SQLERR    // SQL error occurred on delete
DAX_RET_SQLERR    // SQL error occurred on retrieve
DAX_REF_SQLERR    // SQL error occurred on refresh
DAX_SEL_SQLERR    // SQL error occurred on select
DAX_ADD_NONNULL   // Add with non-nullable column not mapped
DAX_NUL_NONNULL   // Cannot SetToNull non-nullable column
DAX_DFT_READONLY  // Cannot SetReadOnly to false on a readonly table/view
DAX_SYS_LOCK      // Error occurred during system semaphore/locking call
DAX_ADD_NULL_DATAID // Add with a null DataId
DAX_UPD_NULL_DATAID // Update with a null DataId
DAX_DEL_NULL_DATAID // Delete with a null DataId
DAX_RET_NULL_DATAID // Retrieve with a null DataId
```

Constructors

```
public:
IDSAccessError(
      const char* a,
      unsigned long b = 0,
      Severity c = IException::unrecoverable,
      struct sqlca* sqlca_p=0
      );
```

You can create objects of this class by doing the following:

- Using the constructor.

errorText	The text describing this particular error.
errorId	The identifier you want to associate with this particular error.
severity	Use the enumeration IException::Severity (see page 386) to specify the severity of the error. The default is unrecoverable.
sqlca	Uses the contents of the sqlca at the time of the exception.

Public Members

name

```
virtual const char *name() const;
```

Returns the name of the object's class.

getSqlca

```
struct sqlca const& getSqlca() const;
```

Returns the contents of the sqlca at the time of the exception.

Inherited Public Members

For information on the members inherited from the IException class, see “IException” on page 379.

Data Access Builder SOM Classes

This chapter describes the SOM version of the Database Access classes. Use these classes when you want to use SOM objects.

Datastore & DatastoreFactory

Datastore and DatastoreFactory provide client connection to the database, disconnect from the database, and commit and rollback of transactions.

For additional information, see “IDatastore” on page 549.

```
interface DatastoreFactory : SOMClass {

    Datastore create_object();
    Datastore create_object_defaults(in string datastore_name,
                                     in string user_name,
                                     in string authentication);

    #ifdef __SOMIDL__
    implementation {
        releaseorder :
                      create_object;
                      create_object_defaults;
    };
    #endif
};

interface Datastore : SOMObject {
  void
      connect   ( in string datastore_name,
                  in string user_name,
                  in string authentication,
      raises    (DaxExcep::ConnectFailed,
                 DaxExcep::DatastoreConnectionInUse,
                 DaxExcep::DatastoreAccessError,
                 DaxExcep::DatastoreLogonFailed,
                 DaxExcep::DatastoreLogoffFailed);

  void
      connect_defaults ()
      raises    (DaxExcep::ConnectFailed,
                 DaxExcep::DatastoreConnectionInUse,
                 DaxExcep::DatastoreAccessError,
                 DaxExcep::DatastoreLogonFailed,
                 DaxExcep::DatastoreLogoffFailed);
  void
      disconnect ()
      raises     (DaxExcep::DatastoreConnectionNotOpen,
                  DaxExcep::DisconnectError,
                  DaxExcep::DatastoreLogoffFailed);

  void
      commit ()
      raises        DaxExcep::DatastoreAccessError,
                     DaxExcep::DatastoreConnectionNotOpen);

  void
      rollback ()
      raises        DaxExcep::DatastoreAccessError,
                     DaxExcep::DatastoreConnectionNotOpen);

  boolean
      is_connected ();
  boolean
      is_sharemode_exclusive();
  void
      enable_sharemode_exclusive (in boolean enable);

  attribute string datastore_name;
  attribute string user_name;
  attribute string authentication;

   #ifdef __SOMIDL__
   implementation
   {
      metaclass = DatastoreFactory;

      releaseorder: connect,
                    connect_defaults,
                    disconnect,
                    commit,
                    rollback,
                    is_connected,
                    is_sharemode_exclusive,
                    enable_sharemode_exclusive,
                    _get_datastore_name,
                    _set_datastore_name,
                    _get_user_name,
                    _set_user_name,
                    _get_authentication,
                    _set_authentication,

      datastore_name: noset, noget;
      user_name: noset, noget;
      authentication: noset, noget;

      somInit: override;
      somUninit: override;
   };
   #endif
};
#endif
```

PersistentObject & POFactory

The PersistentObject class provides the basic data manipulation operations where a client can call directly to add, update, delete or retrieve a row from a table. It is the abstract base class for all of the parts generated by the tool. For additional information, see "IPersistentObject" on page 553.

The POFactory class provides operations to deal with collections of rows from a table. For additional information, see "IPOManager" on page 556.

```
interface POFactory : SOMClass
{
   exception POFError
   {
      long error_code;
      long sqlcode;
   };

   sequence<PersistentObject> retrieveAll() raises(POFError);
   sequence<PersistentObject> select(in string clause) raises(POFError);

   void setPOFException(in long errorCode, in long sqlcode);

   #ifdef __SOMIDL__
   implementation
   {
      releaseorder : retrieveAll,
                     select,
                     setPOFException;
   };
   #endif
};
interface PersistentObject : SOMObject
{
   exception POError
   {
      long error_code;
      long sqlcode;
   };

   void add() raises(POError);
   void update() raises(POError);
   void del() raises(POError);
   void retrieve() raises(POError);
   void setPOException(in long errorCode, in long sqlcode);

   #ifdef __SOMIDL__
   implementation
   {
      releaseorder : add,
                     update,
                     del,
                     retrieve,
                     setPOException;
       metaclass = POFactory;
   };
   #endif
};
```

When an exception occurs, error_code is set with the following values:

```
DAX_ADD_READONLY  // Add used on a readonly table/view
DAX_ADD_SQLERR    // SQL error occurred on add
DAX_UPD_READONLY  // Update used on a readonly table/view
DAX_UPD_SQLERR    // SQL error occurred on update
DAX_DEL_READONLY  // Delete used ona readonly table/view
DAX_DEL_SQLERR    // SQL error occurred on delete
DAX_RET_SQLERR    // SQL error occurred on retrieve
DAX_REF_SQLERR    // SQL error occurred on refresh
DAX_SEL_SQLERR    // SQL error occurred on select
DAX_ADD_NONNULL   // Add with non-nullable column not mapped
DAX_NUL_NONNULL   // Cannot SetToNull non-nullable column
DAX_DFT_READONLY  // Cannot SetReadOnly to false on a readonly table/view
DAX_SYS_LOCK      // Error occurred during system semaphore/locking call
DAX_ADD_NULL_DATAID  // Add with a null DataId
DAX_UPD_NULL_DATAID  // Update with a null DataId
DAX_DEL_NULL_DATAID  // Delete with a null DataId
DAX_RET_NULL_DATAID  // Retrieve with a null DataId
```

If the exception is an SQL exception, the sqlcode is set with the SQLCODE returned from the static SQL statement.

Part 9. Appendix, Bibliography, Glossary, and Index

Header Files for Collection Class Library Coding Examples

This appendix contains edited header files used by some coding examples found in this book. The following header files are shown:

Header File	Page	Header File	Page
animal.h	575	parcel.h	582
circle.h	576	planet.h	583
curve.h	577	toyword.h	584
demoelem.h	578	transelm.h	584
dsur.h	579	trmapops.h	585
line.h	581	xebc2asc.h	586
graph.h	580		

These header files can be found in `...\ibmclass\samples\iclcc`. Some comments and white space have been removed.

animal.h

```
// animal.h  -  Class Animal for use with the example animals.C

  #include <iglobals.h>           // For definition of Boolean
  #include <istring.hpp>          // Class IString
  #include <iostream.h>

class Animal {
  IString nm;
  IString attr;

public:

  Animal(IString n, IString a) : nm(n), attr(a)  {}

  // For copy constructor we use the compiler generated default.
  // For assignment we use the compiler generated default.

  IBoolean operator==(Animal const& p) const  {
     return  ((nm == p.name()) && (attr == p.attribute()));
  }

  IString const& name() const {
     return nm;
  }

  IString const& attribute() const {
     return attr;
  }

  friend ostream& operator<<(ostream& os, Animal const& p)  {
     return os << "The " << p.name() << " is " << p.attribute()
     << "." << endl;
  }
};

  // Key access:
inline IString const& key(Animal const& p)  {
  return p.name();
}

  // We need a hash function for the key type as well.
  // Let's just use the default provided for IString.
inline unsigned long hash(Animal const& animal, unsigned long n) {
  return hash(animal.name(), n);
}
```

circle.h

```
// circle.h

#include <istring.hpp>

class Circle : public Graphics {
public:
  float ivXCenter;
  float ivYCenter;
  float ivRadius;

  Circle(int graphicsKey, IString id ,
         double xCenter, double yCenter,
         double radius)
                          : Graphics(graphicsKey, id),
                            ivXCenter(xCenter),
                            ivYCenter(yCenter),
                            ivRadius(radius) { }

  IBoolean operator== (Circle const& circle) const {
     return (this->ivXCenter == circle.ivXCenter &&
             this->ivYCenter == circle.ivYCenter &&
             this->ivRadius == circle.ivRadius);
    }


  void         draw() const {
     cout << "drawing "
          << Graphics::id() << endl
          << "with center: "
          << "(" << this->ivXCenter << "|"
          << this->ivYCenter << ")"
          << " and with radius: "
          << this->ivRadius << endl;
    }

  void         circumference() const {
     cout << "The circumference of "
          << Graphics::id() << " is: "
          << ((this->ivRadius)*2*3.14) << endl;
    }
};
```

curve.h

```
// curve.h

#include <istring.hpp>

class Curve : public Graphics {
public:

  float ivXStart, ivYStart;
  float ivXFix1, ivYFix1;
  float ivXFix2, ivYFix2;
  float ivXFix3, ivYFix3;
  float ivXEnd,  ivYEnd;

  Curve(int graphicsKey, IString id,
        float xstart, float ystart,
        float xfix1, float yfix1,
        float xfix2, float yfix2,
        float xfix3, float yfix3,
        float xend, float yend)  :
  Graphics(graphicsKey, id),
  ivXStart(xstart), ivYStart(ystart),
  ivXFix1(xfix1),   ivYFix1(yfix1),
  ivXFix2(xfix2),   ivYFix2(yfix2),
  ivXFix3(xfix3),   ivYFix3(yfix3),
  ivXEnd(xend),     ivYEnd(yend) { }

  IBoolean operator== (Curve const& curve) const {
     return (this->ivXStart == curve.ivXStart &&
             this->ivYStart == curve.ivYStart &&
             this->ivXFix1  == curve.ivXFix1  &&
             this->ivYFix1  == curve.ivYFix1  &&
             this->ivXFix2  == curve.ivXFix2  &&
             this->ivYFix2  == curve.ivYFix2  &&
             this->ivXFix3  == curve.ivXFix3  &&
             this->ivYFix3  == curve.ivYFix3  &&
             this->ivXEnd   == curve.ivXEnd   &&
             this->ivYEnd   == curve.ivYEnd);
    }

  void           draw() const {
     cout << "drawing " << Graphics::id()
          << "\nwith starting point: "
          << "(" << this->ivXStart << "|"
          << this->ivYStart << ")"
          << " and with fix points: "
          << "(" << this->ivXFix1 << "|" << this->ivYFix1 << ")"
          << "(" << this->ivXFix2 << "|" << this->ivYFix2 << ")"
          << "(" << this->ivXFix3 << "|" << this->ivYFix3 << ")\n"
          << "and with ending point: "
          << "(" << this->ivXEnd << "|" << this->ivYEnd << ")"
          << endl;
    }
  void          lengthOfCurve() const {
     cout << "Length of "
          << Graphics::id()
          << " is: "
          << (sqrt(pow(((this->ivXFix1) - (this->ivXStart)),2)
                + pow(((this->ivYFix1) - (this->ivYStart)),2))
            + sqrt(pow(((this->ivXFix2) - (this->ivXFix1)),2)
                + pow(((this->ivYFix2) - (this->ivYFix1)),2))
            + sqrt(pow(((this->ivXFix3) - (this->ivXFix2)),2)
                + pow(((this->ivYFix3) - (this->ivYFix2)),2))
            + sqrt(pow(((this->ivXEnd) -  (this->ivXFix3)),2)
                + pow(((this->ivYEnd) - (this->ivYFix3)),2)))
          << endl;
    }
};
```

demoelem.h

```
// demoelem.h  -  DemoElement for use with Key Collections
#ifndef _DEMOELEM_H
#define _DEMOELEM_H

#include <stdlib.h>
#include <iglobals.h>
#include <iostream.h>
#include <istdops.h>

class DemoElement {
  int i, j;

public:
           DemoElement () : i(0), j(0) {}
           DemoElement (int i,int j) : i (i), j(j) {}
           operator int () const { return i; }

  IBoolean operator == (DemoElement const& k) const
           { return i == k.i && j == k.j; }

  IBoolean operator < (DemoElement const& k) const
           { return i < k.i || (i == k.i && j < k.j); }

  friend unsigned long hash (DemoElement const& k, unsigned long n)
           { return k.i % n; }

  int const & geti () const { return i; }
  int const & getj () const { return j; }
};

inline ostream & operator << (ostream &sout, DemoElement const& e)
{ sout << e.geti () << "," << e.getj ();
  return sout;
}

inline int const& key (DemoElement const& k) { return k.geti (); }

// NOTE: You must return a const & in the key function!  Otherwise the
// standard element operations will return a reference to a temporary.
// This would lead to incorrect behavior of the collection operations.

// The key function must be declared in the header file of
// the collection's element type.

// If either of these is not possible or is undesirable,
// an element operations class must be used.
#endif
```

dsur.h

```
// dsur.h  -  Class for Disk Space Usage Records
//            Used by Sorted Map and Sorted Relation example
  #include <fstream.h>
  #include <string.h>
  #include <iglobals.h>
  const int bufSize = 62;

  class DiskSpaceUR    {
     int       blocks;
     char*     name;

   public:
     DiskSpaceUR() {}
     DiskSpaceUR (DiskSpaceUR const& dsur)  { init(dsur); }
     void operator= (DiskSpaceUR const& dsur)   {
       deInit();
       init(dsur);
     }

     DiskSpaceUR (istream& DSURfile) { DSURfile >> *this; }
     ~DiskSpaceUR () { deInit(); }

     IBoolean operator == (DiskSpaceUR const& dsur) const  {
       return (blocks == dsur.blocks)
           && strcmp (name, dsur.name) == 0;
     }


     friend istream& operator >> (istream& DSURfile,
                                  DiskSpaceUR& dsur)     {
         DSURfile >> dsur.blocks;

         char temp[bufSize];
         DSURfile.get(temp, bufSize);

         if (DSURfile.good())  {
                              // Remove leading tabs and blanks
            for (int cnt=0;
                 (temp[cnt] == '\t') || (temp[cnt] == ' ');
                 cnt++) {}
            dsur.name = new char[strlen(temp+cnt)+1];
            strcpy(dsur.name, temp+cnt);
         }
         else   {
            dsur.setInvalid();
            dsur.name = new char[1];
            dsur.name[0] = '\0';
         }

         return DSURfile;
     }

     friend ostream& operator << (ostream& outstream,
                                  DiskSpaceUR& dsur)     {
         outstream.width(bufSize);
         outstream.setf(ios::left, ios::adjustfield);
         outstream << dsur.name;

         outstream.width(9);
         outstream.setf(ios::right, ios::adjustfield);
         outstream << dsur.blocks;

         return outstream;
     }
     inline int const& space () const {return blocks;}
     inline char* const& id () const {return name;}
     inline IBoolean isValid () const {return (blocks > 0);}

   protected:

     inline void init (DiskSpaceUR const& dsur)     {
        blocks = dsur.blocks;
        name = new char[strlen(dsur.name) + 1];
        strcpy(name, dsur.name);
     }

     inline void deInit() {  delete[] name;  }
     inline void setInvalid () { blocks = -1;}
  };


     // Key access on name
  inline  char* const& key (DiskSpaceUR const& dsur)  {
     return dsur.id();
  }

     // Key access on space used
     // Since we can not have two key functions with same args
     // in global name space, we need to use an operations class.
  #include <istdops.h>
     // We can inherit all from the default operations class
     // and then define just the key access function ourselves.
     // We cannot use StdKeyOps here, because they in turn
     // use the key function in global name space, which is
     // already defined for keys of type char* above.
  class DSURBySpaceOps :  public IStdMemOps,
                          public IStdAsOps< DiskSpaceUR >,
                          public IStdEqOps< DiskSpaceUR >    {
    public:
       IStdCmpOps < int > keyOps;

     // Key Access
       int const& key (DiskSpaceUR const& dsur) const
       { return dsur.space(); }
  };
```

graph.h

```
#include <istring.hpp>
#include <iostream.h>

class Graphics {
protected:
  IString ivId;    //*** graphics ID ****/
  int     ivKey;   //*** graphics key ****/

public:
  Graphics (int graphicsKey, IString id) : ivKey(graphicsKey),
                                           ivId(id) { }

   Graphics() {
     cout << this->ivId  << " will now be deleted ... "  << endl;
    }

  IBoolean operator== (Graphics const& graphics) const {
     return (this->ivId == graphics.ivId);
    }
  IString const& id() const   { return ivId; }

  virtual void draw() const =0;

  /**** This member function returns the graphic's key ****/
  /*    Note that we are returning the int by reference,  */
  /*    because this member function will be used by the  */
  /*    key(...) function, which must return a reference. */
  /******************************************************/
  int const& graphicsKey() const {
     return ivKey;
    }
};

/****************       key function       *********************/
/****   note that this interface must always be used with:  ****/
/****             Keytype const& key(....)                  ****/
/****   We are providing this key function for the element  ****/
/****   type Graphics  and not for the managed pointer.     ****/
/**************************************************************/
  inline int const& key (Graphics const& graphics) {
     return graphics.graphicsKey();
    }
```

line.h

```
#include <istring.hpp>
#include <math.h>

class Line : public Graphics {
public:

  double ivXStart, ivYStart;
  double ivXEnd, ivYEnd;

  Line(int graphicsKey, IString id, double xstart, double ystart,
                                  double xend,   double yend) :

     Graphics(graphicsKey, id),
     ivXStart(xstart), ivYStart(ystart),
     ivXEnd(xend), ivYEnd(yend) { }

  IBoolean operator== (Line const& line) const {
     return (this->ivXStart == line.ivXStart &&
             this->ivYStart == line.ivYStart &&
             this->ivXEnd == line.ivXEnd &&
             this->ivYEnd == line.ivYEnd);
    }

  void          draw() const {
     cout << "drawing " << Graphics::id() << endl
          << "with starting point: "
          << "(" << this->ivXStart << "|" << this->ivYStart << ")"
          << " and with ending point: "
          << "(" << this->ivXEnd   << "|" << this->ivYEnd << ")" << endl;
    }

  void          lengthOfLine() const {
     cout << "The length of line " << Graphics::id() << " is: "
          << sqrt(pow(((this->ivXEnd) - (this->ivXStart)),2)
                + pow(((this->ivYEnd) - (this->ivYStart)),2))
          << endl;
    }
};
```

Example Header Files

parcel.h

```
// parcel.h  -  Class Parcel and its parts for use with the
//              example for Key Sorted Set and Heap.
#include <iostream.h>

                        // For definition of Boolean:
#include <iglobals.h>
                        // Class IString:
#include <istring.hpp>

class PlaceTime {

  IString cty;
  int   daynum;  // Keeping it simple:  January 9 is day 9

public:
  PlaceTime(IString acity, int aday) : cty(acity), daynum(aday) {}
  PlaceTime(IString acity) : cty(acity) {daynum = 0;}
  IString const& city() const { return cty; }
  int const& day() const { return daynum; }
  void operator=(PlaceTime const& pt) {
      cty = pt.cty;
      daynum = pt.daynum;
  }

  IBoolean operator==(PlaceTime const& pt)  const {
     return ( (cty == pt.cty)
           && (daynum == pt.daynum) );
  }
};


class Parcel {
  PlaceTime org, lstAr;
  IString dst, id;

public:

  Parcel(IString orig,  IString dest, int day, IString ident)
     :  org(orig, day),  lstAr(orig, day),  dst(dest), id(ident) {}

  void arrivedAt(IString const& acity, int const& day) {
     PlaceTime nowAt(acity, day);
                              // Only if not already there before
     if (nowAt.city() != lstAr.city())
        lstAr = nowAt;
  }

  void wasDelivered(int const& day) {arrivedAt(dst, day);  }
  PlaceTime const& origin() const { return org; }
  IString const& destination() const { return dst; }
  PlaceTime const& lastArrival() const { return lstAr; }
  IString const& ID() const { return id; }

  friend ostream& operator<<(ostream& os, Parcel const& p) {
    os << p.id << ": From " << p.org.city()
       << "(day "  << p.org.day() << ") to " << p.dst;

    if (p.lstAr.city() != p.dst) {
       os << endl << "           is at " << p.lstAr.city()
          << "  since day " << p.lstAr.day() << ".";
    }
    else {
       os << endl << "           was delivered on day "
          << p.lstAr.day() << ".";
    }
    return os;
  }
};

                        // Key access:
  inline  IString const& key( Parcel const&  p) { return p.ID(); }
                        // We need a compare function for the key.
                        // Let's use the default provided for IString:
  inline long compare(Parcel const& p1, Parcel const& p2) {
     return compare(p1.ID(), p2.ID());
  }
```

planet.h

```
// planet.h  -  Class Planet for use in our Sorted Set example

class Planet   {
 private:
   char* plname;
   float dist, mass, bright;

 public:
     // Use the compiler generated default for the copy constructor

   Planet(char* aname, float adist, float amass, float abright) :
     plname(aname), dist(adist),  mass(amass), bright(abright) {}

     // For any Set we need to provide element equality.
   IBoolean operator== (Planet const& aPlanet) const
      { return plname == aPlanet.plname; }

     // For a Sorted Set we need to provide element comparision.
   IBoolean operator< (Planet const& aPlanet) const
      { return dist < aPlanet.dist; }

   char*   name()     { return  plname; }

   IBoolean isHeavy()  { return (mass   > 1.0); }
   IBoolean isBright() { return (bright < 0.0); }
};


// Iterator
#include <iostream.h>

class SayPlanetName : public IIterator<Planet>   {
 public:
   virtual IBoolean applyTo(Planet& p)
         { cout << " " << p.name() << " "; return True;}
};
```

Example Header Files

toyword.h

```
// toyword.h  -  Class Word for use with coding examples.

#include <istring.hpp>

class Word  {
        IString       ivWord;
        unsigned      ivKey;

public:

  //Constructor to be used for sample: wordbag.c
  Word (IString word, unsigned theLength) : ivWord(word),
                                            ivKey(theLength)
                                            {}

  //Constructor to be used for sample: wordseq.c
  Word (IString word) : ivWord(word) {}

  IBoolean operator> (Word const& w1) {
            return this->ivWord > w1.ivWord;
            }

  unsigned setKey() {
            this->ivKey = this->ivWord.length();
            return this->ivKey;
            }

  IString const& getWord() const { return this->ivWord; }
  unsigned const& getKey() const { return this->ivKey; }
};

       // Key access.  The length of the word is the key.
inline unsigned const& key (Word const &aWord)
{ return aWord.getKey(); }
```

transelm.h

```
// transelm.h  -  For use with the Translation Table example.
#ifndef _TRANSELM_H
#define _TRANSELM_H

#include <iglobals.h>

class TranslationElement  {

  char ivAscCode;
  char ivEbcCode;

public:

  /* Let the compiler generate Default and Copy Constructor,*/
  /* Destructor and Assignment for us.                      */

  char const& ascCode () const { return ivAscCode; }
  char const& ebcCode () const { return ivEbcCode; }

  TranslationElement (char asc, char ebc)
       : ivAscCode(asc), ivEbcCode(ebc) {};
  /* We need to define the equality relation.              */
  IBoolean operator == (TranslationElement const& te) const  {
     return ivAscCode == te.ivAscCode
        &&  ivEbcCode == te.ivEbcCode;
   };

  /* An ordering relation must not be defined for          */
  /* elements in a map.                                    */

  /* We need to define the key access for the elements.    */
  /* We decided to define all key operations in a          */
  /* separate operations class in file trmapops.h.         */

};
#endif
```

trmapops.h

```
// trmapops.h  -  Translation Map Operations
//  Base class for element operations for Translation Map example
#ifndef _TRMAPOPS_H
#define _TRMAPOPS_H

// Get the standard operation classes.
#include <istdops.h>

#include "transelm.h"

class TranslationOps : public IEOps < TranslationElement >
{
public:
  class KeyOps : public IStdEqOps < char >, public IStdHshOps < char >
  {
  }  keyOps;
};

// Operations Class for the EBCDIC-ASCII mapping:
class TranslationOpsE2A : public TranslationOps
{
public:              // Key Access
  char const& key (TranslationElement const& te) const
    { return te.ebcCode (); }
};

// Operations Class for the ASCII-EBCDIC mapping:
class TranslationOpsA2E : public TranslationOps
{
public:              // Key Access
  char const& key (TranslationElement const& te) const
    { return te.ascCode (); }
};
#endif
```

xebc2asc.h

```
// xebc2asc.h :   EBCDIC - ASCII Translation Table.
 const unsigned char translationTable[256] = {
0x00,0x01,0x02,0x03,0xCF,0x09,0xD3,0x7F,0xD4,0xD5,0xC3,0x0B,0x0C,0x0D,0x0E,0x0F,
0x10,0x11,0x12,0x13,0xC7,0xB4,0x08,0xC9,0x18,0x19,0xCC,0xCD,0x83,0x1D,0xD2,0x1F,
0x81,0x82,0x1C,0x84,0x86,0x0A,0x17,0x1B,0x89,0x91,0x92,0x95,0xA2,0x05,0x06,0x07,
0xE0,0xEE,0x16,0xE5,0xD0,0x1E,0xEA,0x04,0x8A,0xF6,0xC6,0xC2,0x14,0x15,0xC1,0x1A,
0x20,0xA6,0xE1,0x80,0xEB,0x90,0x9F,0xE2,0xAB,0x8B,0x9B,0x2E,0x3C,0x28,0x2B,0x7C,
0x26,0xA9,0xAA,0x9C,0xDB,0xA5,0x99,0xE3,0xA8,0x9E,0x21,0x24,0x2A,0x29,0x3B,0x5E,
0x2D,0x2F,0xDF,0xDC,0x9A,0xDD,0xDE,0x98,0x9D,0xAC,0xBA,0x2C,0x25,0x5F,0x3E,0x3F,
0xD7,0x88,0x94,0xB0,0xB1,0xB2,0xFC,0xD6,0xFB,0x60,0x3A,0x23,0x40,0x27,0x3D,0x22,
0xF8,0x61,0x62,0x63,0x64,0x65,0x66,0x67,0x68,0x69,0x96,0xA4,0xF3,0xAF,0xAE,0xC5,
0x8C,0x6A,0x6B,0x6C,0x6D,0x6E,0x6F,0x70,0x71,0x72,0x97,0x87,0xCE,0x93,0xF1,0xFE,
0xC8,0x7E,0x73,0x74,0x75,0x76,0x77,0x78,0x79,0x7A,0xEF,0xC0,0xDA,0x5B,0xF2,0xF9,
0xB5,0xB6,0xFD,0xB7,0xB8,0xB9,0xE6,0xBB,0xBC,0xBD,0x8D,0xD9,0xBF,0x5D,0xD8,0xC4,
0x7B,0x41,0x42,0x43,0x44,0x45,0x46,0x47,0x48,0x49,0xCB,0xCA,0xBE,0xE8,0xEC,0xED,
0x7D,0x4A,0x4B,0x4C,0x4D,0x4E,0x4F,0x50,0x51,0x52,0xA1,0xAD,0xF5,0xF4,0xA3,0x8F,
0x5C,0xE7,0x53,0x54,0x55,0x56,0x57,0x58,0x59,0x5A,0xA0,0x85,0x8E,0xE9,0xE4,0xD1,
0x30,0x31,0x32,0x33,0x34,0x35,0x36,0x37,0x38,0x39,0xB3,0xF7,0xF0,0xFA,0xA7,0xFF
};
```

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 Reference
Volume I
Version 3.0

Publication No. S25H-6965-00

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 Reference
Volume I
Version 3.0

Publication No. S25H-6965-00

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

Name	Address
Company or Organization	
Phone No.	

Readers' Comments — We'd Like to Hear from You
S25H-6965-00

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-6965-00

Cut or Fold
Along Line

IBM®

Part Number: 25H6965
Program Number: 30H1664
30H1665
30H1666

Printed in U.S.A.

25H6965

S25H-6965-00