

IBM VisualAge for C++ for Windows

S33H-5039-00

Open Class Library Reference
Volume I

Version 3.5



IBM VisualAge for C++ for Windows

S33H-5039-00

**Open Class Library Reference
Volume I**

Version 3.5

Note!

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

First Edition (February 1996)

This edition applies to 3.5 of IBM VisualAge for C++ (Programs 33H4979 and 33H4980) 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.

© Copyright International Business Machines Corporation 1992, 1996. All rights reserved.

Note to U.S. Government Users — Documentation related to restricted rights — Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corp.

Contents

| | |
|---|-----------|
| About This Book | 1 |
| <hr/> | |
| Part 1. Complex Mathematics Library | 7 |
| complex Class | 8 |
| c_exception Class | 16 |
| <hr/> | |
| Part 2. I/O Stream Library | 21 |
| filebuf Class | 23 |
| fstream, ifstream, and ofstream Classes | 27 |
| ios Class | 34 |
| iostream and iostream_withassign Classes | 48 |
| istream and istream_withassign Classes | 50 |
| Manipulators | 60 |
| ostream and ostream_withassign Classes | 63 |
| stdiobuf and stdiostream Classes | 71 |
| streambuf Class | 74 |
| strstream, istrstream, and ostrstream Classes | 87 |
| strstreambuf Class | 91 |
| <hr/> | |
| Part 3. Flat Collection Classes | 95 |
| Introduction to Flat Collections | 96 |
| Flat Collection Member Functions | 99 |

| | |
|--|-----|
| Bag | 131 |
| Deque | 136 |
| Equality Sequence | 142 |
| Heap | 146 |
| Key Bag | 149 |
| Key Set | 155 |
| Key Sorted Bag | 162 |
| Key Sorted Set | 168 |
| Map | 176 |
| Priority Queue | 184 |
| Queue | 188 |
| Relation | 192 |
| Sequence | 195 |
| Set | 201 |
| Sorted Bag | 207 |
| Sorted Map | 212 |
| Sorted Relation | 219 |
| Sorted Set | 224 |
| Stack | 231 |
| <hr/> | |
| Part 4. Tree Collection Classes | 237 |
| Introduction to Trees | 238 |
| Multiway Tree | 240 |

| | |
|---|-----|
| Part 5. Auxiliary Collection Classes | 257 |
| Cursor | 258 |
| Tree Cursor | 261 |
| Applicator and Constant Applicator Classes | 265 |
| Pointer Classes | 266 |
| Part 6. Abstract Collection Classes | 271 |
| Collection | 272 |
| Equality Collection | 273 |
| Equality Key Collection | 274 |
| Equality Key Sorted Collection | 275 |
| Equality Sorted Collection | 276 |
| Key Collection | 277 |
| Key Sorted Collection | 278 |
| Ordered Collection | 279 |
| Sequential Collection | 280 |
| Sorted Collection | 281 |
| Part 7. Data Type, Stream, and Exception Classes | 283 |
| Class Hierarchy | 286 |
| I0String | 287 |
| IAccessError | 304 |
| IAssertionFailure | 306 |

| | |
|--------------------------------------|-----|
| IBase | 308 |
| IBase::Version | 313 |
| IBaseErrorInfo | 314 |
| IBaseStream | 320 |
| IBitFlag | 336 |
| IBuffer | 340 |
| ICLibErrorInfo | 358 |
| IContext | 362 |
| IDate | 365 |
| IDBCSBuffer | 376 |
| IDeviceError | 392 |
| IException | 394 |
| IException::TraceFn | 404 |
| IExceptionLocation | 406 |
| IFileStream | 408 |
| IGUIErrorInfo | 413 |
| IInvalidParameter | 418 |
| IInvalidRequest | 420 |
| IMemoryStream | 422 |
| IMessageText | 426 |
| IMetaType | 429 |
| IMetaTypeInfo | 432 |

| | |
|---------------------------------|-----|
| INotificationEvent | 435 |
| INotifier | 439 |
| IObserver | 444 |
| IObserverList | 447 |
| IObserverList::Cursor | 450 |
| IOutOfMemory | 453 |
| IOutOfSystemResource | 455 |
| IOutOfWindowResource | 457 |
| IPair | 459 |
| IPoint | 467 |
| IPointArray | 470 |
| IRange | 474 |
| IRectangle | 477 |
| IRefCounted | 492 |
| IReference | 495 |
| IResourceExhausted | 498 |
| ISize | 500 |
| IStandardNotifier | 503 |
| IString | 508 |
| IStringEnum | 561 |
| IStringParser | 562 |
| IStringParser::SkipWords | 571 |

| | |
|--------------------------------------|-----|
| IStringTest | 573 |
| IStringTestMemberFn | 577 |
| ISystemErrorInfo | 580 |
| ITime | 584 |
| ITrace | 590 |
| IVBase | 598 |
| IXLibErrorInfo | 601 |

Part 8. Data Access Builder C++ Classes and Exception

| | |
|------------------------------------|-----|
| Classes | 605 |
| IDAException | 607 |
| IDatastore | 610 |
| IDatastoreBase | 616 |
| IDatastoreDB2 | 624 |
| IDatastoreODBC | 632 |
| IPersistentObject | 640 |
| IPOManager | 645 |

Part 9. Data Access Builder SOM Classes and Exceptions . . . 647

| | |
|--------------------------------------|-----|
| Datastore | 648 |
| DatastoreBase | 650 |
| DatastoreDB2 | 655 |
| DatastoreDB2Factory | 660 |
| DatastoreFactory | 661 |

| | |
|--|-----|
| DatastoreODBC | 662 |
| DatastoreODBCFactory | 667 |
| PersistentObject | 668 |
| POFactory | 670 |
| <hr/> | |
| Part 10. Appendix, Glossary, Bibliography, and Index | 673 |
| Appendix A. Header Files for Collection Class Library Coding Examples | 674 |
| Appendix B. Glossary | 686 |
| Appendix C. Bibliography | 718 |
| Appendix D. Index | 719 |



Notices

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

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the

IBM Director of Licensing,
IBM Corporation,
500 Columbus Avenue,
Thornwood, NY, 10594
USA.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independent created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact IBM Canada Ltd., Department 071, 1150 Eglinton Avenue East, North York, Ontario M3C 1H7, Canada. Such information may be available, subject to appropriate terms and conditions, including in some cases payment of a fee.

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

Trademarks and Service Marks

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

C Set ++

CUA

IBMLink

System Object Model

WorkFrame

Common User Access

IBM

OS/2

VisualAge

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

Windows is a trademark of Microsoft Corporation.

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

IBM's VisualAge products and services are not associated with or sponsored by Visual Edge Software Ltd..



About This Book

This book describes the classes and class members of some of the C++ class libraries that are part of IBM Open Class Library, the comprehensive set of class libraries provided with VisualAge for 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 Builder 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

About Examples

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 99, 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 User Interface Class Library Examples

User Interface Class Library class and member descriptions may contain links to samples illustrating those classes and members. Up to five samples are listed for each class/member. These samples are identified at the end of the panel, or, for overloaded functions, after each overload. When you click on such links, an LPEX editor session starts up, loads the sample program from the `\ibmcpp\samples\ioc` directory, and positions the cursor at the first instance of the member or class being illustrated.

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 `...\ibmcpp\samples\ioc`. 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 674 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.


Related Books



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

Related Documentation

See Appendix C, “Bibliography” on page 718 for a list of related books and suggested reading materials.

How to Get Help

There are three kinds of online information available to you while you are using VisualAge for C++:

Online documents

These are complete documents, like the one you are reading now, presented online. These documents contain detailed information on the different aspects of VisualAge for C++. For your convenience, the online documents are presented in:

- Standard format (.INF files). See “Getting Help Inside VisualAge for C++” on page 4 for instructions on opening standard format documents from inside VisualAge for C++. See “Getting Help from the Command Line” on page 4 for instructions on opening standard format documents from the command line. For a list of the VisualAge for C++ documents that are available in standard format, see “Online Documents Available in VisualAge for C++” on page 5.

Contextual help

Contextual help is available throughout VisualAge for C++. This help tells you all about the elements that you see in the interface, including menus, entry fields, and pushbuttons.

How Do I help

Many of the common tasks that you want to perform with VisualAge for C++ are described in *How Do I* help. The *How Do I* help for a task gives you step-by-step instructions for completing the task. There is overall *How Do I* help for VisualAge for C++, as well as individual task lists for each of its components.

Related Books

Getting Help Inside VisualAge for C++

All three kinds of help are available directly within the VisualAge for C++ interface:

- To get general contextual help for the component of VisualAge for C++ that you are using, press **F1** anywhere in the window.
- To get contextual help on a particular menu, menu item, or button, highlight the element and press **F1**.
- To get access to all of the help information that is available to you in a particular window, click on **Help** in the menu bar at the top of the window. This menu includes the following selections:
 - **Help Index**, an alphabetical list of all of the help topics that are available from this window
 - **General Help**, overall help for the window
 - **Using Help**, general information about the help facility
 - **How Do I...**, the How Do I help for the component
 - **Product Information**, a dialog that shows the level of VisualAge for C++ being used

In addition, there are selections that let you open all of online documents that are available in VisualAge for C++.

- To get detailed information, open the **Online Information** notebook in the VisualAge for C++ folder. In this notebook you will find tabs for **Guides**, **References**, and **How Do I** help. Each page in the notebook lists a variety of online documents that describe, in detail, the different aspects of VisualAge for C++. To open a particular online document, select the radio button for the document, and click on the **View** pushbutton.

Getting Help from the Command Line

If you want, you can look at the online documents by issuing the `iview` command.

The installation routine stores the online document files in the `\IBMCPW\HELP` directory. To view the *Language Reference*, for example, make `C:\IBMCPW\HELP` your current directory (substituting the drive where you installed VisualAge for C++ for `C:`) and enter the following command:

```
IVIEW CPPLNG.INF
```

If you want to get information on a specific topic, you can specify a word or a series of words after the file name. If the words appear in an entry in the table of contents or the index, the online document is opened to the associated section. For example, if you want to read the section on operator precedence in the *Language Reference*, you can enter the following command:

```
IVIEW CPPLNG.INF OPERATOR PRECEDENCE
```

Related Books

Getting Help for a Keyword or Construct

If you are editing a file using the Editor, you can get help for a keyword or construct by moving the cursor to the word and pressing **Ctrl+H**. In the other tools, you can get help for a keyword or construct by highlighting the word and pressing **Ctrl+H**.

Online Documents Available in VisualAge for C++

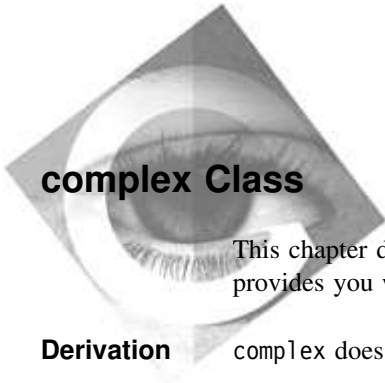
The following documents are available in standard format:

| | |
|--|--|
| <i>Building VisualAge for C++ Parts for Fun and Profit</i> | <i>Open Class Library Reference</i> |
| <i>C Library Reference</i> | <i>Open Class Library User's Guide</i> |
| <i>Editor Command Reference</i> | <i>Programming Guide</i> |
| <i>Frequently Asked Questions</i> | <i>SOM Programming Guide</i> |
| <i>Installation Guide and Product Overview</i> | <i>SOM Programming Reference</i> |
| <i>IPF User's Guide</i> | <i>User's Guide</i> |
| <i>IPF Programmer's Guide and Reference</i> | <i>Visual Builder User's Guide</i> |
| <i>Language Reference</i> | <i>Visual Builder Parts Reference</i> |

Related Books

Part 1. Complex Mathematics Library

| | |
|---|--------|
| complex Class | 8 |
| Constants Defined in complex.h | 8 |
| Constructors for complex | 9 |
| Mathematical Operators for complex | 10 |
| Input and Output Operators for complex | 12 |
| Mathematical Functions for complex | 12 |
| Trigonometric Functions for complex | 13 |
| Magnitude Functions for complex | 14 |
| Conversion Functions for complex | 14 |
| c_exception Class | 16 |
| Constructor for c_exception | 16 |
| Data Members of c_exception | 16 |
| Errors Handled by the Complex Mathematics Library | 17 |
| Errors Not Handled by the Complex Mathematics Library | 19 |



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 | 9 | <code>conj</code> | 14 |
| operator <code>+</code> | 10 | <code>cos</code> | 13 |
| operator <code>+=</code> , <code>-=</code> , <code>*=</code> , <code>/=</code> | 11 | <code>cosh</code> | 13 |
| operator <code>!=</code> | 11 | <code>exp</code> | 12 |
| operator <code>*</code> | 10 | <code>imag</code> | 15 |
| operator <code>-</code> (negation) | 10 | <code>log</code> | 12 |
| operator <code>-</code> (subtraction) | 10 | <code>norm</code> | 14 |
| operator <code>/</code> | 11 | <code>polar</code> | 15 |
| operator <code>>></code> | 12 | <code>pow</code> | 13 |
| operator <code><<</code> | 12 | <code>real</code> | 15 |
| operator <code>==</code> | 11 | <code>sin</code> | 14 |
| <code>abs</code> | 14 | <code>sinh</code> | 14 |
| <code>arg</code> | 14 | <code>sqrt</code> | 13 |

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):

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

| Constant Name | Description |
|-----------------------|--|
| <code>M_E</code> | The constant e |
| <code>M_LOG2E</code> | The logarithm of e to the base of 2 |
| <code>M_LOG10E</code> | The logarithm of e to the base of 10 |
| <code>M_LN2</code> | The natural logarithm of 2 |
| <code>M_LN10</code> | The natural logarithm of 10 |
| <code>M_PI</code> | π |
| <code>M_PI_2</code> | $\pi / 2$ |

complex Constructors

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

| Constant Name | Description |
|---------------|---------------------------------------|
| M_PI_4 | $\pi / 4$ |
| M_1_PI | $1 / \pi$ |
| M_2_PI | $2 / \pi$ |
| 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.

complex Mathematical Operators

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 .

complex Input and Output

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.  Figure 2 on page 18 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 .  Figure 2 on page 18 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 .  Figure 2 on page 18 shows the values returned by the default error-handling procedure for **cosh()**.

complex Magnitude Functions

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*.  Figure 2 on page 18 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*);

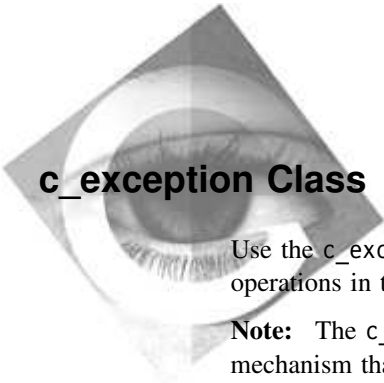
arg() returns the angle (in radians) of the polar representation of its argument. If the argument *x* is equal to the complex number (*a*,*b*), the angle returned is the angle in radians on the complex plane between the real axis and the vector (*a*,*b*). The return value has a range of $-\pi$ to π .  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 | <pre>friend complex polar(double a, double $b=0$);</pre> <p><code>polar()</code> returns the standard complex representation of the complex number that has a polar representation (a,b).</p> |
| real | <pre>friend double real(const complex& x);</pre> <p><code>real()</code> extracts the real part of the complex number x.</p> |
| imag | <pre>friend double imag(const complex& x);</pre> <p><code>imag()</code> extracts the imaginary part of the complex number x.</p> |



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 | 16 | name | 16 |
| arg1 | 16 | retval | 17 |
| arg2 | 16 | type | 17 |

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 of the error handling function `complex_error()` returns. You can make your own definition of `complex_error()` to 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 link 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`.

Figure 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

Figure 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 `CPPWRTM.DLL`. The message number in `CPPWRTM.DLL` 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 for C++ for Windows User's Guide*.

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()`.

Errors Not Handled by the Complex Library

Part 2. I/O Stream Library

| | |
|---|----|
| filebuf Class | 23 |
| Public Members of filebuf | 24 |
| fstream, ifstream, and ofstream Classes | 27 |
| Public Members of fstreambase | 27 |
| Public Members offstream | 28 |
| Public Members of ifstream | 30 |
| Public Members of ofstream | 32 |
| ios Class | 34 |
| Constructors and Assignment Operator for ios | 35 |
| Format State Variables | 35 |
| Format State Flags | 36 |
| Public Members of ios for the Format State | 39 |
| Public Members of ios for User-Defined Format Flags | 42 |
| Public Members of ios for the Error State | 43 |
| Other Members of ios | 45 |
| Built-In Manipulators for ios | 47 |
| iostream and iostream_withassign Classes | 48 |
| Public Members of iostream and iostream_withassign | 48 |
| istream and istream_withassign Classes | 50 |
| Constructors for istream | 50 |
| Input Prefix Function | 51 |
| Public Members of istream for Formatted Input | 51 |
| Public Members of istream for Unformatted Input | 55 |
| Public Members of istream for Positioning | 57 |
| Other Public Members of istream | 57 |
| Built-In Manipulators for istream | 58 |
| Public Members of istream_withassign | 59 |
| Manipulators | 60 |
| Parameterized Manipulators for the Format State | 60 |
| ostream and ostream_withassign Classes | 63 |
| Constructors for ostream | 63 |
| Output Prefix and Suffix Functions | 64 |
| Public Members of ostream for Formatted Output | 64 |

| | |
|--|----|
| Public Members of ostream for Unformatted Output | 68 |
| Public Members of ostream for Positioning | 68 |
| Other Public Members of ostream | 69 |
| Built-In Manipulators for ostream | 69 |
| Public Members of ostream_withassign | 70 |
| stdiobuf and stdiostream Classes | 71 |
| Public Members of stdiobuf | 71 |
| Public Members of stdiostream | 72 |
| streambuf Class | 74 |
| streambuf Public and Protected Interfaces | 74 |
| Public Members of the streambuf Public Interface | 76 |
| Protected Functions That Return Pointers | 78 |
| Protected Functions That Set Pointers | 80 |
| Other Nonvirtual Protected Member Functions | 81 |
| Protected Virtual Member Functions | 83 |
| strstream, istrstream, and ostrstream Classes | 87 |
| Public Members of strstreambase | 87 |
| Public Members of strstream | 88 |
| Public Members of istrstream | 89 |
| Public Members of ostrstream | 90 |
| strstreambuf Class | 91 |
| Public Members of strstreambuf | 91 |



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` object 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 |
|----------------------------------|------|----------------------|------|
| <code>filebuf</code> constructor | 24 | <code>is_open</code> | 25 |
| <code>filebuf</code> destructor | 24 | <code>open</code> | 25 |
| <code>attach</code> | 24 | <code>seekoff</code> | 25 |
| <code>close</code> | 24 | <code>seekpos</code> | 26 |
| <code>detach</code> | 24 | <code>setbuf</code> | 26 |
| <code>fd</code> | 25 | <code>sync</code> | 26 |

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*.

filebuf Public Members

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*.

Note: This member is not supported under C++/MVS.

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.

Note: This member is not supported under C++/MVS.

close

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

close() does the following:

1. Flushes any output that is waiting in *fb* to be sent to the file

filebuf Public Members

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.

Note: This member is not supported under C++/MVS.

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*.

filebuf Public Members

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 | 28 | constructor | 30 |
| close | 28 | open | 31 |
| detach | 28 | rdbuf | 31 |
| setbuf | 28 | ofstream: | |
| fstream: | | constructor | 32 |
| constructor | 28 | open | 32 |
| open | 29 | rdbuf | 33 |
| rdbuf | 30 | | |

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*.

fstream

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*.

Note: This member function is not supported under C++/MVS.

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()`.

Note: This member function is not supported under C++/MVS.

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 ifstream

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

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 *fs*.

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

fstream

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 ORed 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 | <code>open()</code> 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 | <code>open()</code> performs a seek to the end of the file. Setting <code>ios::ate</code> does not open the file for input or output. If you set <code>ios::ate</code> , you should explicitly set <code>ios::in</code> , <code>ios::out</code> , or both. |
| ios::bin | See <code>ios::binary</code> below. |

ifstream

| | |
|-----------------------|---|
| 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 <code>ios::binary</code> , 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. <code>ios::binary</code> and <code>ios::bin</code> 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. <code>ios::noreplace</code> is ignored if <code>ios::in</code> 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 <code>ios::out</code> and neither <code>ios::ate</code> nor <code>ios::app</code> , you are implicitly specifying <code>ios::trunc</code> . If you set <code>ios::trunc</code> , you should explicitly set <code>ios::in</code> , <code>ios::out</code> , or both. |
| ios::nocreate | If the file does not exist, the call to <code>open()</code> fails. |
| ios::noreplace | If the file already exists and <code>ios::out</code> is set, the call to <code>open()</code> fails. If <code>ios::out</code> is not set, <code>ios::noreplace</code> 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 fildesc);
```

ifstream

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 ORed 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 29 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`.

ofstream

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);
```

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*.

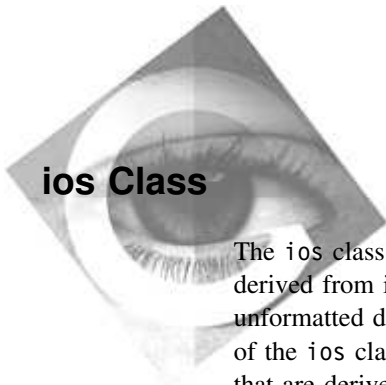
ofstream

The members of the `ios::open_mode` enumeration are bits that can be ORed 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 29 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 |
|--------------------------------|------|------------------------------|------|
| <code>ios</code> constructor | 35 | <code>precision</code> | 40 |
| <code>bad</code> | 43 | <code>pword</code> | 42 |
| <code>bitalloc</code> | 42 | <code>rdbuf</code> | 45 |
| <code>clear</code> | 43 | <code>rdstate</code> | 44 |
| <i>dec</i> | 37 | <i>right</i> | 37 |
| <code>dec</code> manipulator | 47 | <i>scientific</i> | 38 |
| <code>endl</code> manipulator | 47 | <code>setf</code> | 40 |
| <code>ends</code> manipulator | 47 | <i>showbase</i> | 37 |
| <code>eof</code> | 43 | <i>showpoint</i> | 38 |
| <code>fail</code> | 44 | <i>showpos</i> | 37 |
| <code>fill</code> | 39 | <code>skip</code> | 41 |
| <i>fixed</i> | 38 | <i>skipws</i> | 36 |
| <code>flags</code> | 40 | <i>stdio</i> | 39 |
| <code>flush</code> manipulator | 47 | <code>sync_with_stdio</code> | 45 |
| <code>good</code> | 44 | <code>tie</code> | 46 |
| <i>hex</i> | 37 | <i>unitbuf</i> | 39 |
| <code>hex</code> manipulator | 47 | <code>unsetf</code> | 41 |
| <i>internal</i> | 37 | <i>uppercase</i> | 38 |
| <code>isword</code> | 42 | <code>width</code> | 41 |
| <i>left</i> | 37 | <code>ws</code> manipulator | 47 |
| <i>oct</i> | 37 | <i>x_fill</i> | 35 |
| <code>oct</code> manipulator | 47 | <i>x_precision</i> | 36 |
| <code>operator void*</code> | 44 | <i>x_width</i> | 36 |
| <code>operator=</code> | 35 | <code>xalloc</code> | 42 |

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 59 and “Assignment Operator for `ostream_withassign`” on page 70 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.

Format State Flags

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 39 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 47 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.

Format State Flags

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 45 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 36 for more details.

You can also use the parameterized manipulator `setfill` to set the value of `ios::x_fill`.  See “`setfill`” on page 61 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 62 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 62 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

Format State Members

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 61 for more details on this parameterized manipulator.

Note: If you set conflicting flags the results are unpredictable.  See “Mutually Exclusive Format Flags” on page 39 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`,

User-Defined Format Flags

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 62 for more information on this parameterized manipulator. Also, see “Public Members of `ostream` for Formatted Output” on page 64 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 36, 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.

word

```
long& word(int i);
```

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

ios Error State

`xalloc()` simply returns a new index; it does not do any allocation. `word()` and `pword()` do the allocation, and if the allocation fails, they set `ios::failbit`. You should check `ios::failbit` after calling `word()` 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 35 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 39 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.

Other ios Members

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:

| | |
|--------------------|---|
| <code>dec</code> | <code>istream</code> and <code>ostream</code> |
| <code>hex</code> | <code>istream</code> and <code>ostream</code> |
| <code>oct</code> | <code>istream</code> and <code>ostream</code> |
| <code>ws</code> | <code>istream</code> |
| <code>endl</code> | <code>ostream</code> |
| <code>ends</code> | <code>ostream</code> |
| <code>flush</code> | <code>ostream</code> |

 See “Built-In Manipulators for `istream`” on page 58 for more details on the built-in manipulators for `istream`. See “Built-In Manipulators for `ostream`” on page 69 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.
- `stringstream`, 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 |
|--|------|
| <code>iostream</code> Constructor | 48 |
| <code>iostream_withassign</code> Constructor | 48 |
| <code>iostream_withassign</code> Assignment Operator | 49 |

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 35 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.

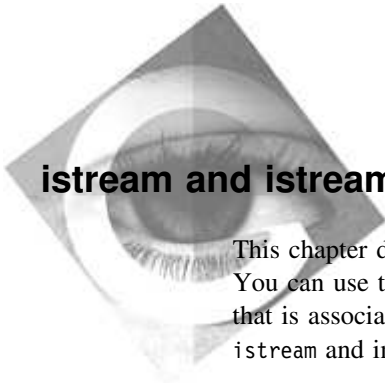
iostream and iostream_withassign Classes

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 |
|----------------------------------|------|---|------|
| <code>ipfx</code> | 51 | <code>tellg</code> | 57 |
| <code>istream</code> Constructor | 50 | <code>gcount</code> | 57 |
| <code>input</code> operator | 51 | <code>peek</code> | 58 |
| <code>get</code> | 55 | <code>putback</code> | 58 |
| <code>getline</code> | 56 | <code>sync</code> | 58 |
| <code>ignore</code> | 56 | <code>istream_withassign</code> Constructor | 59 |
| <code>read</code> | 57 | <code>istream_withassign</code> operator= | 59 |
| <code>seekg</code> | 57 | | |

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 35 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 35 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 52 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 *rc*. 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);  
istream& operator>>(long long& ir);  
istream& operator>>(unsigned long long& ir);
```

Note: The support for `long long` is controlled by the `-q(no)longlong` option. By default, `long long` is supported.

This section describes how the input operator works for references to the integral types: `short`, `unsigned short`, `int`, `unsigned int`, `long`, `unsigned long`, `long long`, and `unsigned long 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.

Formatted Input

- `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 -12 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);
```

Unformatted Input

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 51 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*.

Unformatted Input

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 55 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 51 for more details on formatted input.)

Built-In Manipulators

peek `int peek();`

`peek()` calls `ipfx(1)`. If `ipfx()` returns zero, or if no more input is available from the ultimate producer, `peek()` returns `E0F`. 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 51 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 `E0F`.

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`:

istream_withassign

```
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();
```

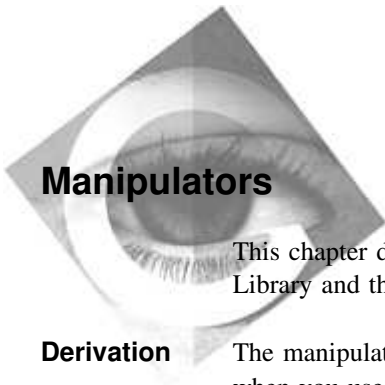
The `istream_withassign` constructor creates an `istream_withassign` object. It does not do any initialization of this object.

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” in the *Open Class Library User's Guide* for further information.

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

Members The following parameterized manipulators are described:

| Manipulator | Page | Manipulator | Page |
|----------------------------|------|---------------------------|------|
| <code>resetiosflags</code> | 61 | <code>setiosflags</code> | 62 |
| <code>setbase</code> | 61 | <code>setprecision</code> | 62 |
| <code>setfill</code> | 61 | <code>setw</code> | 62 |

Parameterized Manipulators for the Format State

The `omanip.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 36 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`.”

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


Manipulators

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 40 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 39 for more details on `fill()`.

Manipulators

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 40 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 40 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 41 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);
```



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 | 63 | osfx | 64 |
| output operator | 65 | put | 68 |
| ostream_withassign constructor | 70 | seekp | 68 |
| ostream_withassign operator= | 70 | tellp | 69 |
| flush | 69 | write | 68 |
| opfx | 64 | | |

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 35 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

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 36) 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.

Formatted Output

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 support for long long is controlled by the `-q(no)longlong` option. By default, long long is supported.

For the integral types (short, unsigned short, int, unsigned int, long, unsigned long, long long, and unsigned long 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:

Formatted Output

- `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 39 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 38 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 36 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*.

Unformatted Output

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*.

Other Public Members of ostream

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;
```

ostream_withassign

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();
```

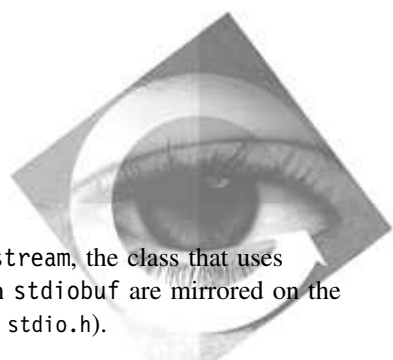
The `ostream_withassign` constructor creates an `ostream_withassign` object. It does not do any initialization on the object.

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 27 and “`filebuf` Class” on page 23 for more details on these classes. See “`sync_with_stdio`” on page 45 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 | 71 | Constructor | 72 |
| Destructor | 72 | <code>rdbuf</code> | 72 |
| <code>stdiofile</code> | 72 | | |

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`.

stdiostream

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);
    }
    //
```

stdiostream

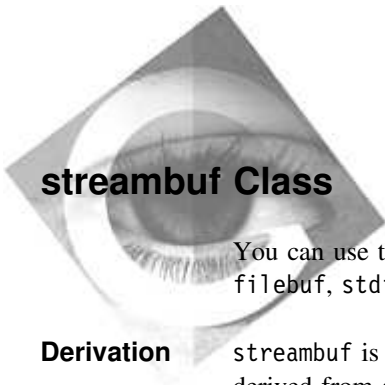
```
// Create a stdiostream object attached to "f"
//
stdiostream sin(in);
stdiostream sout(out);
cout << "The data contained in the file is: " << endl;
//
// Now read data from "sin" and copy it to
// "cout" and "sout"
//
while ((c = sin.rdbuf()->sgetc()) != EOF)
{
    cout << char(c);
    sout.rdbuf()->sputc(c);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



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 |
|-------------------------------------|------|-------------------------|------|
| <code>streambuf</code> constructors | 76 | <code>pptr</code> | 79 |
| <code>streambuf</code> destructor | 76 | <code>sbumpc</code> | 77 |
| <code>allocate</code> | 81 | <code>seekoff</code> | 84 |
| <code>base</code> | 78 | <code>seekpos</code> | 85 |
| <code>blen</code> | 81 | <code>setb</code> | 80 |
| <code>dbp</code> | 81 | <code>setbuf</code> | 85 |
| <code>doallocate</code> | 83 | <code>setg</code> | 80 |
| <code>eback</code> | 78 | <code>setp</code> | 80 |
| <code>ebuf</code> | 79 | <code>sgetc</code> | 77 |
| <code>egptr</code> | 79 | <code>sgetn</code> | 77 |
| <code>epptr</code> | 79 | <code>snextc</code> | 77 |
| <code>gbump</code> | 82 | <code>sputbackc</code> | 77 |
| <code>gptr</code> | 79 | <code>sputc</code> | 78 |
| <code>in_avail</code> | 76 | <code>sputn</code> | 78 |
| <code>out_waiting</code> | 77 | <code>stossc</code> | 78 |
| <code>overflow</code> | 83 | <code>sync</code> | 86 |
| <code>pbackfail</code> | 84 | <code>unbuffered</code> | 82 |
| <code>pbase</code> | 79 | <code>underflow</code> | 86 |
| <code>pbump</code> | 82 | | |

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 Interface?

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

Public Members of streambuf

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 `phase()` 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 86 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.

Functions That Return Pointers

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*.

stoss `void stoss();`

`stoss()` moves the get pointer forward one character. If the get pointer is already at the end of the get area, `stoss()` 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 58 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()`.

Functions That Set Pointers

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 78.

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 33 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 Member Functions

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 82 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.

Other Nonvirtual Member Functions

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, ep_ptr=0x91410, eback=0, gp_ptr=0, eg_ptr=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 `gp_ptr()` is valid.

The beginning of the get area is equal to the value returned by `gp_ptr()`.  See “gp_ptr” on page 79 for more details on `gp_ptr()`.

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 `pp_ptr()` is valid.

The beginning of the put area is equal to the value returned by `pp_ptr()`.  See “pp_ptr” on page 79 for more details on `pp_ptr()`.

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 81 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 81 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 `E0F` 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 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 82 for more details on `unbuffered()`. See “base” on page 78 for more details on `base()`.

overflow `virtual int overflow(int c = E0F);`

`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 `E0F`. `overflow()` should return `E0F` if an error occurs, and it should return a value not equal to `E0F` 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 78 for more details on `eback()`. See “gptr” on page 79 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.

Virtual Member Functions

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.

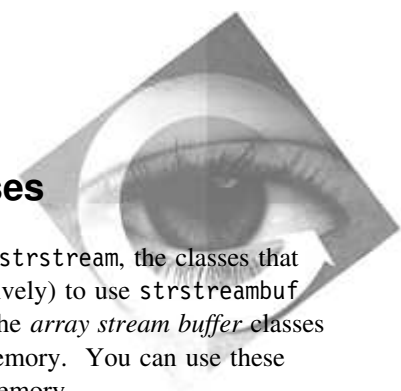
If you define `sync()` in a class derived from `streambuf`, it should send any characters that are stored in the put area to the ultimate consumer, and (if possible) send any characters that are waiting in the get area back to the ultimate producer. When `sync()` returns, both the put area and the get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



stringstream, istream, and ostream Classes

This chapter describes `stringstream`, `ostrstream`, and `stringstream`, the classes that specialize `istream`, `ostream`, and `iostream` (respectively) to use `stringstreambuf` 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 `stringstreambase`, the class from which the array stream buffer classes are derived.

| | |
|------------|--------------|
| Derivation | ios |
| | istream |
| | ostream |
| | iostream |
| | stringstream |
| ios | istream |
| | istream |
| ios | ostream |
| | ostrstream |

Header File `stringstream`, `istream`, and `ostream` are declared in `stream.h`.

Members The following members are provided for `stringstream`, `istream`, and `ostream`:

| Method | Page | Method | Page |
|--------------------------|------|-------------------------|------|
| istream constructors | 89 | stringstream destructor | 88 |
| istream destructor | 89 | pcount | 90 |
| ostream constructors | 90 | rdbuf | 88 |
| ostream destructor | 90 | str (stringstream) | 88 |
| stringstream constructor | 88 | str (ostream) | 90 |

Public Members of stringstreambase

Note: The `stringstreambase` class is an internal class that provides common functions for the classes that are derived from it. Do not use the `stringstreambase` class directly. The following description is provided so that you can use the function as part of `istream`, `ostrstream`, and `stringstream` objects.

stringstream

rdbuf `stringstreambuf* rdbuf();`

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

Public Members of stringstream

Constructor for stringstream

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

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

The version of the `stringstream` 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 57 for more details on `seekg()`.

Destructor for stringstream

```
~stringstream();
```

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

str `char* str();`

`str()` returns a pointer to the stream buffer attached to the `stringstream` 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 `stringstream` 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 `istream`

Constructors for `istream`

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

The versions of the `istream` 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 57 for more details on `seekg()`.

The versions of the `istream` 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 `istream`

```
~istream();
```

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

ostream

Public Members of ostream

Constructors for ostream

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

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

The versions of the `ostream` constructor that take three arguments specify that the stream buffer that is attached to the `ostream` 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 57 for more details on `seekg()`.

Destructor for ostream

```
~ostream();
```

The `ostream` destructor frees space allocated by the `ostream` 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 `ostream` 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 `ostream` 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 `ostream` 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 |
|--|------|------------------------|------|
| <code>strstreambuf</code> constructors | 91 | <code>seekoff</code> | 93 |
| <code>strstreambuf</code> destructors | 92 | <code>setbuf</code> | 94 |
| <code>doallocate</code> | 92 | <code>str</code> | 93 |
| <code>freeze</code> | 93 | <code>underflow</code> | 94 |
| <code>overflow</code> | 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.

strstreambuf Class

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

strstreambuf Class

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

strstreambuf Class

external representation. For further details, see “Low-Level I/O” in the *IBM VisualAge for C++ for Windows C Library Reference*, S33H-5038.

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

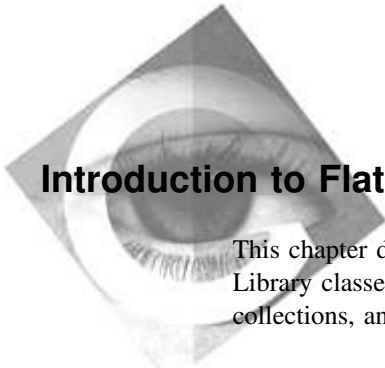
This part contains detailed descriptions of the flat Collection Classes.



“Introduction to Flat Collections” on page 96 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 97.

| | |
|--|-----|
| Introduction to Flat Collections | 96 |
| Flat Collection Member Functions | 99 |
| Bag | 131 |
| Deque | 136 |
| Equality Sequence | 142 |
| Heap | 146 |
| Key Bag | 149 |
| Key Set | 155 |
| Key Sorted Bag | 162 |
| Key Sorted Set | 168 |
| Map | 176 |
| Priority Queue | 184 |
| Queue | 188 |
| Relation | 192 |
| Sequence | 195 |
| Set | 201 |
| Sorted Bag | 207 |
| Sorted Map | 212 |
| Sorted Relation | 219 |
| Sorted Set | 224 |
| Stack | 231 |



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.

Format of Class Descriptions

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.
 - 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:

Types Defined for the Collection Class Library

- 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  “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`:

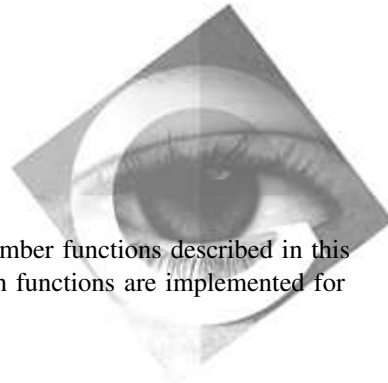
```
typedef int IBoolean;

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
```

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 102.

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!= `IBoollean 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 102.

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== `IBoollean 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 <i>n</i> occurrences in one collection, it must have exactly <i>n</i> occurrences in the other collection. |
| Sequential | The ordering of the elements is the same for both collections. |

Flat Collection Member Functions

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 110 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

Flat Collection Member Functions

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.

Exceptions

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

Flat Collection Member Functions

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

Flat Collection Member Functions

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.
- $(1 \leq position \leq numberOfElements + 1)$.
- 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 112.

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, $(numberOfElements() < maxNumberOfElements())$.

Flat Collection Member Functions

- 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 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

Flat Collection Member Functions

addOrReplaceElementWithKey

```
IBoollean addOrReplaceElementWithKey (  
    Element const& element );  
  
IBoollean 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

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 130.

Flat Collection Member Functions

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

```
IBoolen allElementsDo (
    IBoolen (*function) (Element&, void*),
    void* additionalArgument = 0 ) ;

IBoolen allElementsDo (
    IBoolen (*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 123.
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.

Flat Collection Member Functions

Return Value: Returns true if the given function returns true for every element it is applied to.

allElementsDo

```
IBoolean allElementsDo ( IApplicator <Element>& applicator ) ;
```

```
IBoolean allElementsDo ( IConstantApplicator <Element>& applicator ) const;
```

Calls the `applyTo()` function of the given applicator 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 applicator class. (For further details, see “Iteration Using `allElementsDo`” 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 123.
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.

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

Flat Collection Member Functions

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. If you need a comparison based on the `LC_COLLATE` category, then you must implement your own `compare()` function as described in “Using Separate Functions” in the *Open Class Library User's Guide*

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.

Flat Collection Member Functions

containsElementWithKey

IBoollean **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 102.

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.

Flat Collection Member Functions

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 $m \leq n$.

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.

Flat Collection Member Functions

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,
($1 \leq \textit{position} \leq \text{numberOfElements}()$).

Precondition: ($1 \leq \textit{position} \leq \text{numberOfElements}()$).

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.

Flat Collection Member Functions

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 $\text{MIN}(m,n)$ 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.

Flat Collection Member Functions

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`

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.

Flat Collection Member Functions

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`

Flat Collection Member Functions

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`

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`

Flat Collection Member Functions

locateNextElementWithKey

```
IBoollean 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 IBoollean locateOrAdd ( Element const& element ) ;
```

```
IBoollean locateOrAdd ( Element const& element, ICursor& cursor ) ;
```

Locates an element in the collection that is equal to the given element. (See “locate” on page 116 for details on `locate()`.) If no such element is found, `locateOrAdd()` adds the element as described in “add” on page 101. 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.

Flat Collection Member Functions

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

locateOrAddElementWithKey

```
IBoollean locateOrAddElementWithKey (  
    Element const& element ) ;
```

```
IBoollean 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 101. 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

Flat Collection Member Functions

locatePrevious

```
IBoollean 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

```
ICursor* 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.

Flat Collection Member Functions

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.

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

positionAt

IPosition **positionAt** (ICursor const& *cursor*) const;

Determines the position of the current element. Position 1 specifies the first element.

Precondition: The cursor must belong to the collection, and the cursor must point to an element of the collection.

Exception: ICursorInvalidException

Flat Collection Member Functions

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 101), 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 124.

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.

Flat Collection Member Functions

removeAll `INumber removeAll () ;`

Removes all elements from the collection. Element destructors are called as described in “removeAt” on page 124.

Side Effects

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

Return Value: The number of elements removed.

removeAll `INumber removeAll (`
 `IBoolean (*propertyFunction) (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 124.

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.

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” on page 124.

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.

Flat Collection Member Functions

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.”

The first element of the collection has position 1.

Precondition: Position must be a valid position in the collection; that is, $(1 \leq position \leq numberOfElements())$.

Flat Collection Member Functions

Side Effects

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

Exception: IPositionInvalidException

removeElementWithKey

IBoolen **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 124.

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 124.

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

Flat Collection Member Functions

removeLast `void removeLast () ;`

Removes the last element from the collection. Element destructors are called as described in “removeAt” on page 124.

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

`IBoolen replaceElementWithKey (Element const& element) ;`

`IBoolen 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.

Flat Collection Member Functions

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`

reverse

```
reverse  
void reverse ( ) ;
```

Reverses the sequence of elements in the collection.

Side Effects: All cursors of this collection become undefined.

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`

Flat Collection Member Functions

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, $(1 \leq position \leq numberOfElements())$.

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 (<i>element1</i> > <i>element2</i>) |
| 0 | if (<i>element1</i> == <i>element2</i>) |
| <0 | if (<i>element1</i> < <i>element2</i>) |

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.

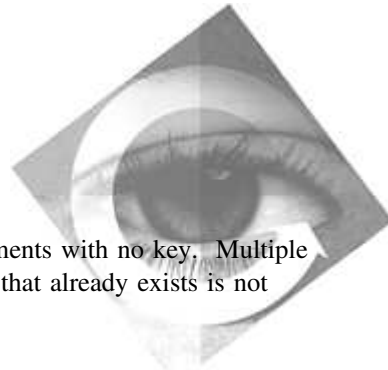
Side Effects

- 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.

 The figure “Combination of Flat Collection Properties” 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 $\text{MIN}(m,n)$ 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 $\leq n$.

Derivation

Collection
Equality Collection
Bag

Variants and Header Files

IBag, the first class in the table below, is the default implementation variant.

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 |
| IBagAsAVLTree | ibagavl.h | AVL tree |
| IGBagAsAVLTree | ibagavl.h | AVL tree |
| IBagAsBstTree | ibagbst.h | B* tree |
| IGBagAsBstTree | ibagbst.h | B* tree |
| IBagAsHshTable | ibaghsh.h | Hash Table |
| IGBagAsHshTable | ibaghsh.h | Hash Table |
| IBagAsList | ibagl1st.h | List |
| IGBagAsList | ibagl1st.h | List |
| IBagAsTable | ibagtab.h | Table |
| IGBagAsTable | ibagtab.h | Table |
| IBagAsDilTable | ibagdil.h | Diluted Table |
| IGBagAsDilTable | ibagdil.h | Diluted Table |


Members

All member functions of flat collections are described in “Introduction to Flat Collections” on page 96. The following members are provided for bag:

| Method | Page | Method | Page |
|------------------|------|---------------------------|------|
| Constructor | 99 | copy | 111 |
| Copy Constructor | 99 | differenceWith | 112 |
| Destructor | 99 | elementAt | 112 |
| operator!= | 100 | intersectionWith | 114 |
| operator= | 100 | isBounded | 115 |
| operator== | 100 | isEmpty | 115 |
| add | 101 | isFull | 115 |
| addAllFrom | 102 | locate | 116 |
| addDifference | 105 | locateNext | 117 |
| addIntersection | 106 | locateOrAdd | 118 |
| addUnion | 107 | maxNumberOfElements | 120 |
| allElementsDo | 108 | newCursor | 120 |
| anyElement | 109 | numberOfDifferentElements | 120 |
| contains | 110 | numberOfElements | 121 |
| containsAllFrom | 110 | numberOfOccurrences | 121 |

Bag

| Method | Page | Method | Page |
|----------------------|------|---------------------------|------|
| remove | 122 | setToFirst | 127 |
| removeAllOccurrences | 124 | setToNext | 128 |
| removeAll | 123 | setToNextDifferentElement | 128 |
| removeAt | 124 | unionWith | 130 |
| replaceAt | 126 | | |

Bag also defines a cursor that inherits from `IElementCursor`.  The members for `IElementCursor` are described in “Cursor” on page 258.

Template Arguments and Required Functions

Bag

`IBag <Element>`
`IGBag <Element, COps>`

The default implementation of the class `IBag` requires the following element functions:

Element Type

- Copy constructor
- Destructor
- Assignment
- Ordering relation

Bag as AVL Tree

`IBagAsAvlTree <Element>`
`IGBagAsAvlTree <Element, COps>`

The implementation of the class `IBagAsAvlTree` requires the following element functions:

- Copy constructor
- Destructor
- Assignment
- Ordering relation

Bag as B* Tree

`IBagAsBstTree <Element>`
`IGBagAsBstTree <Element, COps>`

The implementation of the class `IBagAsBstTree` requires the following element functions:

Bag

Element Type

- Copy constructor
- Destructor
- Assignment
- Ordering relation

Bag as Hash Table

```
IBagAsHshTable <Element>  
IGBagAsHshTable <Element, EHOps>
```

The implementation of the class `IBagAsHshTable` requires the following element functions:

Element Type

- Default constructor
- Copy constructor
- Destructor
- Assignment
- Equality test
- Hash function

Bag as List

```
IBagAsList <Element>  
IGBagAsList <Element, COps>
```

The implementation of the class `IBagAsList` requires the following element functions:

Element Type

- Copy constructor
- Destructor
- Assignment
- Ordering relation

Bag as Table

```
IBagAsTable <Element>  
IGBagAsTable <Element, COps>
```

The implementation of the class `IBagAsTable` requires the following element functions:

Element Type

- Copy constructor
- Destructor

Bag

- Assignment
- Ordering relation

Bag as Diluted Table

IBagAsDilTable <Element>
IGBagAsDilTable <Element, COps>


The implementation of the class IBagAsDilTable requires the following element functions:

Element Type

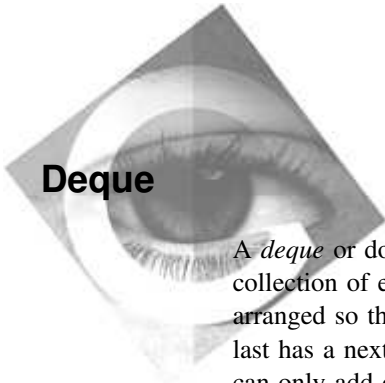
- Copy constructor
- Destructor
- Assignment
- Ordering relation

Abstract Class

IABag <Element>

For polymorphism, you can use the corresponding abstract class, IABag, which is found in the iabag.h header file.  See the section on *Polymorphism and the Collections* in the *Open Class Library User's Guide* for further information.

The required functions are the same as the required functions of the concrete base class.



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.

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 |
|-------------------|-------------|------------------------|
| Deque | idqu.h | List |
| IGDeque | idqu.h | List |
| DequeAsList | idqulst.h | List |
| IGDequeAsList | idqulst.h | List |
| DequeAsTable | idqutab.h | Table |
| IGDequeAsTable | idqutab.h | Table |
| DequeAsDilTable | idqudil.h | Diluted table |
| IGDequeAsDilTable | idqudil.h | Diluted table |

Members

All members of flat collections are described in “Introduction to Flat Collections” on page 96. The following members are provided for deque:

| Method | Page | Method | Page |
|-------------------|------|---------------------|------|
| Constructor | 99 | isFull | 115 |
| Copy Constructor | 99 | isLast | 115 |
| Destructor | 99 | lastElement | 116 |
| operator= | 100 | maxNumberOfElements | 120 |
| add | 101 | newCursor | 120 |
| addAllFrom | 102 | numberOfElements | 121 |
| addAsFirst | 103 | positionAt | 121 |
| addAsLast | 103 | removeAll | 123 |
| allElementsDo | 108 | removeFirst | 125 |
| anyElement | 109 | removeLast | 126 |
| compare | 109 | setToFirst | 127 |
| copy | 111 | setToLast | 127 |
| elementAt | 112 | setToNext | 128 |
| elementAtPosition | 113 | setToPosition | 129 |
| firstElement | 114 | setToPrevious | 129 |
| isBounded | 115 | | |
| isEmpty | 115 | | |
| isFirst | 115 | | |

Deque also defines a cursor that inherits from `IOrderedCursor`.  The members for `IOrderedCursor` are described in “Cursor” on page 258.

Deque

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 as List

```
IDequeAsList <Element>  
IGDequeAsList <Element, StdOps>
```

The implementation variant `IDequeAsList` requires the following element functions:

Element Type

- Copy constructor
- Destructor
- Assignment

Deque as Table

```
IDequeAsTable <Element>  
IGDequeAsTable <Element, StdOps>
```

The implementation of the class `IDequeAsTable` requires the following element functions:

Element Type

- Copy constructor
- Destructor
- Assignment

Deque as Diluted Table

```
IDequeAsDilTable <Element>  
IGDequeAsDilTable <Element, StdOps>
```


The implementation of the class `IDequeAsDilTable` requires the following element functions:

Element Type

- Copy constructor
- Destructor
- Assignment

Abstract Class

IADeque <Element>

For polymorphism, you can use the corresponding abstract class, IADeque, which is found in the `iadqu.h` header file.  See the section on *Polymorphism and the Collections* in the *Open Class Library User's Guide* for further information.

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, ID deque, 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 ID deque <char> Deque;
// The deque requires iteration to be const
typedef IConstantApplicator <char> CharApplicator;

class Print : public CharApplicator
{
public:
    IBoolean applyTo(char const&c)
    {
        cout << c;
        return true;
    }
};

/*-----*\
| Test variables                                |
\*-----*/

char *String = "Teqikbonfxjmsoe aydg.o zlarv pu o wr cu h";
```

Deque

```
/*-----*\
| 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);
}
```

Deque

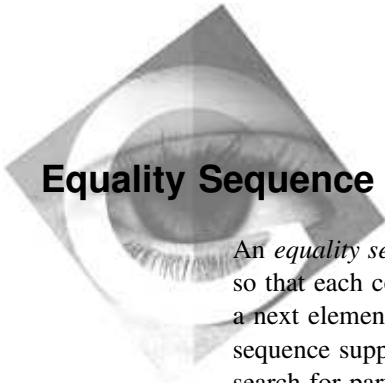
The program produces the following output:

```
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 jumps 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

The figure  “Combination of Flat Collection Properties” 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.

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 | ies.h | List |
| IGEqualitySequence | ies.h | List |
| IEqualitySequenceAsList | ieslst.h | List |
| IGEqualitySequenceAsList | ieslst.h | List |
| IEqualitySequenceAsTable | iestab.h | Table |
| IGEqualitySequenceAsTable | iestab.h | Table |
| IEqualitySequenceAsDilTable | iesdil.h | Diluted table |
| IGEqualitySequenceAsDilTable | iesdil.h | Diluted table |


Members

All members of flat collections are described in “Introduction to Flat Collections” on page 96. The following members are provided for equality sequence:

| Method | Page | Method | Page |
|-------------------|------|----------------------|------|
| Constructor | 99 | isLast | 115 |
| Copy Constructor | 99 | lastElement | 116 |
| Destructor | 99 | locate | 116 |
| operator!= | 100 | locateFirst | 117 |
| operator= | 100 | locateLast | 117 |
| operator== | 100 | locateNext | 117 |
| add | 101 | locateOrAdd | 118 |
| addAllFrom | 102 | locatePrevious | 120 |
| addAsFirst | 103 | maxNumberOfElements | 120 |
| addAsLast | 103 | newCursor | 120 |
| addAsNext | 104 | numberOfElements | 121 |
| addAsPrevious | 104 | numberOfOccurrences | 121 |
| addAtPosition | 105 | positionAt | 121 |
| allElementsDo | 108 | remove | 122 |
| anyElement | 109 | removeAll | 123 |
| compare | 109 | removeAllOccurrences | 124 |
| contains | 110 | removeAt | 124 |
| containsAllFrom | 110 | removeAtPosition | 124 |
| copy | 111 | removeFirst | 125 |
| elementAt | 112 | removeLast | 126 |
| elementAtPosition | 113 | replaceAt | 126 |
| firstElement | 114 | reverse | 127 |
| isBounded | 115 | setToFirst | 127 |
| isEmpty | 115 | setToLast | 127 |
| isFirst | 115 | setToNext | 128 |
| isFull | 115 | setToPosition | 129 |

Equality Sequence

| Method | Page |
|---------------|------|
| setToPrevious | 129 |
| sort | 129 |

Equality sequence also defines a cursor that inherits from `IOrderedCursor`.  The members for `IOrderedCursor` are described in “Cursor” on page 258.

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
- Copy constructor
- Destructor
- Equality test

Equality Sequence as List

`IEqualitySequenceAsList` *<Element>*
`IGEqualitySequenceAsList` *<Element, EOps>*

The implementation of the class `IEqualitySequenceAsList` requires the following element functions:

Element Type

- Assignment
- Copy constructor
- Destructor
- Equality test

Equality Sequence as Table

`IEqualitySequenceAsTable` *<Element>*
`IGEqualitySequenceAsTable` *<Element, EOps>*

The implementation of the class `IEqualitySequenceAsTable` requires the following element functions:

Equality Sequence

Element Type

- Copy constructor
- Destructor
- Assignment
- Equality test

Equality Sequence as Diluted Table

IEqualitySequenceAsDilTable <Element>
IGEqualitySequenceAsDilTable <Element, EOps>


The implementation of the class IEqualitySequenceAsDilTable requires the following element functions:

Element Type

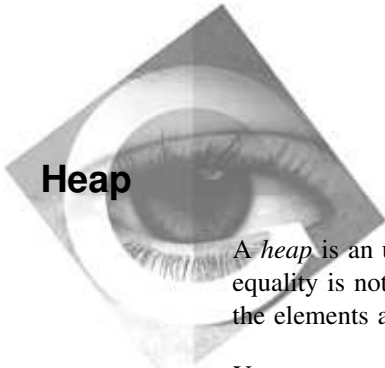
- Copy constructor
- Destructor
- Assignment
- Equality test

Abstract Class

IAEqualitySequence<Element>

For polymorphism, you can use the corresponding abstract class, IAEqualitySequence, which is found in the `iaes.h` header file.  See the section on *Polymorphism and the Collections* in the *Open Class Library User's Guide* for more information.

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.

 The figure “Combination of Flat Collection Properties” 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.

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 | ihp.h | List |
| IGHeap | ihp.h | List |
| IHeapAsList | ihplst.h | List |
| IGHeapAsList | ihplst.h | List |
| IHeapAsTable | ihptab.h | Table |
| IGHeapAsTable | ihptab.h | Table |
| IHeapAsDilTable | ihpdil.h | Diluted table |
| IGHeapAsDilTable | ihpdil.h | Diluted table |

Heap

Members

All members of flat collections are described in “Introduction to Flat Collections” on page 96. The following members are provided for heap:

| Method | Page | Method | Page |
|------------------|------|---------------------|------|
| Constructor | 99 | isEmpty | 115 |
| Copy Constructor | 99 | isFull | 115 |
| Destructor | 99 | maxNumberOfElements | 120 |
| operator= | 100 | newCursor | 120 |
| add | 101 | numberOfElements | 121 |
| addAllFrom | 102 | removeAll | 123 |
| allElementsDo | 108 | removeAt | 124 |
| anyElement | 109 | replaceAt | 126 |
| copy | 111 | setToFirst | 127 |
| elementAt | 112 | setToNext | 128 |
| isBounded | 115 | | |

Heap also defines a cursor that inherits from `IElementCursor`.  The members for `IElementCursor` are described in “Cursor” on page 258.

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 as List

`IHeapAsList` *<Element>*
`IGHeapAsList` *<Element, StdOps>*

The implementation variant `IHeapAsList` requires the following element functions:

Element Type

- Copy constructor
- Assignment

Heap

Heap as Table

```
IHeapAsTable <Element>  
IGHeapAsTable <Element, StdOps>
```

The implementation of the class `IHeapAsTable` requires the following element functions:

Element Type

- Copy constructor
- Assignment

Heap as Diluted Table

```
IHeapAsDilTable <Element>  
IGHeapAsDilTable <Element, StdOps>
```


The implementation of the class `IHeapAsDilTable` requires the following element functions:

Element Type

- Assignment
- Copy constructor


Abstract Class

```
IAHeap<Element>
```

For polymorphism, you can use the corresponding abstract class, `IAHeap`, which is found in the `iahp.h` header file.  See the section on *Polymorphism and the Collections* in the *Open Class Library User's Guide* for further information.

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 172 for an example of using a 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.

 The figure “Behavior of add for Unique and Multiple Collections” 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

 The figure “Combination of Flat Collection Properties” 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.

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 | ikb.h | Hash table |
| IGKeyBag | ikb.h | Hash table |
| IKeyBagAsHshTable | ikbhsh.h | Hash table |
| IGKeyBagAsHshTable | ikbhsh.h | Hash table |

Members

All members of flat collections are described in “Introduction to Flat Collections” on page 96. The following members are provided for key bag:

| Method | Page | Method | Page |
|----------------------------|------|---------------------------|------|
| Constructor | 99 | locateElementWithKey | 116 |
| Copy Constructor | 99 | locateNextElementWithKey | 118 |
| Destructor | 99 | locateOrAddElementWithKey | 119 |
| operator= | 100 | maxNumberOfElements | 120 |
| add | 101 | newCursor | 120 |
| addAllFrom | 102 | numberOfDifferentKeys | 121 |
| addOrReplaceElementWithKey | 107 | numberOfElements | 121 |
| allElementsDo | 108 | numberOfElementsWithKey | 121 |
| anyElement | 109 | removeAll | 123 |
| containsAllKeysFrom | 110 | removeAllElementsWithKey | 123 |
| containsElementWithKey | 111 | removeAt | 124 |
| copy | 111 | removeElementWithKey | 125 |
| elementAt | 112 | replaceAt | 126 |
| elementWithKey | 113 | replaceElementWithKey | 126 |
| isBounded | 115 | setToFirst | 127 |
| isEmpty | 115 | setToNext | 128 |
| isFull | 115 | setToNextWithDifferentKey | 128 |
| key | 115 | | |

Key Bag also defines a cursor that inherits from `IElementCursor`.  The members for `IElementCursor` are described in “Cursor” on page 258.

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:

Key Bag

Element Type

- Copy constructor
- Destructor
- Assignment
- Key access

Key Type

- Equality test
- Hash function

Key Bag as Hash Table

IKeyBagAsHshTable <Element, Key>
IGKeyBagAsHshTable <Element, Key, KEHops>

The implementation of the class IKeyBagAsHshTable requires the following element and key-type functions:

Element Type


- Copy constructor
- Destructor
- Assignment
- Key access

Key Type

- Equality test
- Hash function

Abstract Class

IAKeyBag<Element, Key>

For polymorphism, you can use the corresponding abstract class, IAKeyBag, which is found in the iakb.h header file.  See the section on *Polymorphism and the Collections* in the *Open Class Library User's Guide* for further information.

The required functions are the same as the required functions of the concrete base class.


Key Bag

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 forICursor 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 674 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;
    forICursor(animalsCur1) cout << "    " << animalsCur1.element();
```

Key Bag

```
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:
forICursor(extinctCur1)
    animals.removeAllElementsWithKey(extinctCur1.element());

    // Display all elements in animals:
cout << endl << endl
    << "After removing all observations on extinct animals:" << endl;
forICursor(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;
}
```

Key Bag

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.

 The figure “Behavior of add for Unique and Multiple Collections” 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.

 The figure “Combination of Flat Collection Properties” 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.

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 |
|--------------------------------|-----------------------|------------------------|
| <code>IKeySet</code> | <code>iks.h</code> | AVL tree |
| <code>IGKeySet</code> | <code>iks.h</code> | AVL tree |
| <code>IKeySetAsAvlTree</code> | <code>iksavl.h</code> | AVL tree |
| <code>IGKeySetAsAvlTree</code> | <code>iksavl.h</code> | AVL tree |
| <code>IKeySetAsBstTree</code> | <code>iksbst.h</code> | B* tree |
| <code>IGKeySetAsBstTree</code> | <code>iksbst.h</code> | B* tree |


Key Set

| Class Name | Header File | Implementation Variant |
|--------------------|-------------|------------------------|
| IKeySetAsHshTable | ikshsh.h | Hash table |
| IGKeySetAsHshTable | ikshsh.h | Hash table |
| IKeySetAsList | ikslst.h | List |
| IGKeySetAsList | ikslst.h | List |
| IKeySetAsTable | ikstab.h | Table |
| IGKeySetAsTable | ikstab.h | Table |
| IKeySetAsDilTable | iksdil.h | Diluted table |
| IGKeySetAsDilTable | iksdil.h | Diluted table |

Members

All members of flat collections are described in “Introduction to Flat Collections” on page 96. The following members are provided for key set:

| Method | Page | Method | Page |
|----------------------------|------|---------------------------|------|
| Constructor | 99 | isEmpty | 115 |
| Copy Constructor | 99 | isFull | 115 |
| Destructor | 99 | key | 115 |
| operator= | 100 | locateElementWithKey | 116 |
| add | 101 | locateOrAddElementWithKey | 119 |
| addAllFrom | 102 | maxNumberOfElements | 120 |
| addOrReplaceElementWithKey | 107 | newCursor | 120 |
| allElementsDo | 108 | numberOfElements | 121 |
| anyElement | 109 | removeAll | 123 |
| containsAllKeysFrom | 110 | removeAt | 124 |
| containsElementWithKey | 111 | removeElementWithKey | 125 |
| copy | 111 | replaceAt | 126 |
| elementAt | 112 | replaceElementWithKey | 126 |
| elementWithKey | 113 | setToFirst | 127 |
| isBounded | 115 | setToNext | 128 |

Key set also defines a cursor that inherits from `IElementCursor`.  The members for `IElementCursor` are described in “Cursor” on page 258.

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 as AVL Tree

IKeySetAsAvlTree <*Element*, *Key*>
 IGKeySetAsAvlTree <*Element*, *Key*, *KCOps*>

The implementation of the class IKeySetAsAvlTree requires the following element and key-type functions:

Element Type

- Copy constructor
- Destructor
- Assignment
- Key access

Key Type

Ordering relation

Key Set as B* Tree

IKeySetAsBstTree <*Element*, *Key*>
 IGKeySetAsBstTree <*Element*, *Key*, *KCOps*>

The implementation of the class IKeySetAsBstTree requires the following element and key-type functions:

Key Set

Element Type

- Default constructor
- Copy constructor
- Destructor
- Assignment
- Key access

Key Type

Ordering relation

Key Set as Hash Table

IKeySetAsHshTable <Element, Key>

IGKeySetAsHshTable <Element, Key, KEHOps>

The implementation class IKeySetAsHshTable requires the following element and key-type functions:

Element Type

- Copy constructor
- Destructor
- Assignment
- Key access

Key Type

- Equality test
- Hash function

Key Set as List

IKeySetAsList <Element, Key>

IGKeySetAsList <Element, Key, KCOps>

The implementation of the class IKeySetAsList requires the following element and key-type functions:

Key Set

Element Type

- Copy constructor
- Destructor
- Assignment
- Key access

Key Type

Ordering relation

Key Set as Table

IKeySetAsTable <Element, Key>
IGKeySetAsTable <Element, Key, KCOps>

The implementation of the class IKeySetAsTable requires the following element and key-type functions:

Element Type

- Copy constructor
- Destructor
- Assignment
- Key access

Key Type

Ordering relation

Key Set as Diluted Table

IKeySetAsDilTable <Element, Key>
IGKeySetAsDilTable <Element, Key, KCOps>

The implementation of the class IKeySetAsDilTable requires the following element and key-type functions:

Element Type

- Copy constructor
- Destructor
- Assignment
- Key access


Key Type

Ordering relation

Key Set

Abstract Class

IAKeySet<Element,Key>


For polymorphism, you can use the corresponding abstract class, IAKeySet, which is found in the iaks.h header file.  See the section on *Polymorphism and the Collections* in the *Open Class Library User's Guide* for further information.

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.

 See Appendix A, “Header Files for Collection Class Library Coding Examples” on page 674 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));
    }
```

Key Set

```
        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

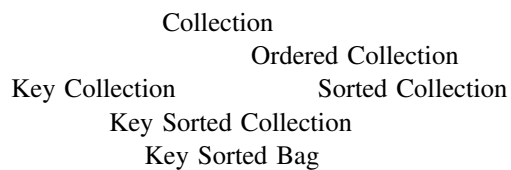
A *key sorted bag* is an ordered collection of zero or more elements that have a key. Elements are sorted according to the value of their key field. Element equality is not supported. Multiple elements are supported.



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.

 The figure “Combination of Flat Collection Properties” 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



Variants and Header Files

IKeySortedBag, the first class in the table below, is the default implementation variant.

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.

Key Sorted Bag


| Class Name | Header File | Implementation Variant |
|--------------------------|-------------|------------------------|
| IKeySortedBag | iksb.h | List |
| IGKeySortedBag | iksb.h | List |
| IKeySortedBagAsList | iksblst.h | List |
| IGKeySortedBagAsList | iksblst.h | List |
| IKeySortedBagAsTable | iksbt.h | Table |
| IGKeySortedBagAsTable | iksbt.h | Table |
| IKeySortedBagAsDilTable | iksbdil.h | Diluted table |
| IGKeySortedBagAsDilTable | iksbdil.h | Diluted table |

Members

All members of flat collections are described in “Introduction to Flat Collections” on page 96. The following members are provided for key sorted bag:

| Method | Page | Method | Page |
|----------------------------|------|---------------------------|------|
| Constructor | 99 | locateElementWithKey | 116 |
| Copy Constructor | 99 | locateNextElementWithKey | 118 |
| Destructor | 99 | locateOrAddElementWithKey | 119 |
| operator= | 100 | maxNumberOfElements | 120 |
| add | 101 | newCursor | 120 |
| addAllFrom | 102 | numberOfDifferentKeys | 121 |
| addOrReplaceElementWithKey | 107 | numberOfElements | 121 |
| allElementsDo | 108 | numberOfElementsWithKey | 121 |
| anyElement | 109 | positionAt | 121 |
| compare | 109 | removeAll | 123 |
| containsAllKeysFrom | 110 | removeAllElementsWithKey | 123 |
| containsElementWithKey | 111 | removeAt | 124 |
| copy | 111 | removeAtPosition | 124 |
| elementAt | 112 | removeElementWithKey | 125 |
| elementAtPosition | 113 | removeFirst | 125 |
| elementWithKey | 113 | removeLast | 126 |
| firstElement | 114 | replaceAt | 126 |
| isBounded | 115 | replaceElementWithKey | 126 |
| isEmpty | 115 | setToFirst | 127 |
| isFirst | 115 | setToLast | 127 |
| isFull | 115 | setToNext | 128 |
| isLast | 115 | setToNextWithDifferentKey | 128 |
| key | 115 | setToPosition | 129 |
| lastElement | 116 | setToPrevious | 129 |

Key Sorted Bag

Key sorted bag also defines a cursor that inherits from `IOrderedCursor`.  The members for `IOrderedCursor` are described in “Cursor” on page 258.

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 as List

`IKeySortedBagAsList` *<Element, Key>*
`IGKeySortedBagAsList` *<Element, Key, KCOps>*

The implementation of the class `IKeySortedBagAsList` requires the following element and key-type functions:

Element Type

- Copy constructor
- Destructor
- Assignment
- Key access

Key Type

Ordering relation

Key Sorted Bag as Table

`IKeySortedBagAsTable` *<Element, Key>*
`IGKeySortedBagAsTable` *<Element, Key, KCOps>*

Key Sorted Bag

The implementation of the class `IKeySortedBagAsTable` requires the following element and key-type functions:

Element Type

- Copy constructor
- Destructor
- Assignment
- Key access

Key Type

Ordering relation

Key Sorted Bag as Diluted Table

`IKeySortedBagAsDilTable` *<Element, Key>*
`IGKeySortedBagAsDilTable` *<Element, Key, KCOps>*

The implementation of the class `IKeySortedBagAsDilTable` requires the following element and key-type functions:

Element Type


- Copy constructor
- Destructor
- Assignment
- Key access

Key Type

Ordering relation

Abstract Class

`IAKeySortedBag<Element, Key>`

For polymorphism, you can use the corresponding abstract class, `IAKeySortedBag`, which is found in the `iaksb.h` header file.  See the section on *Polymorphism and the Collections* in the *Open Class Library User's Guide* for further information.


The required functions are the same as the required functions of the concrete base class.

Key Sorted Bag

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 674 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 <iksb.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);
    for(Cursor(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)
```

Key Sorted Bag

```
        << " 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.

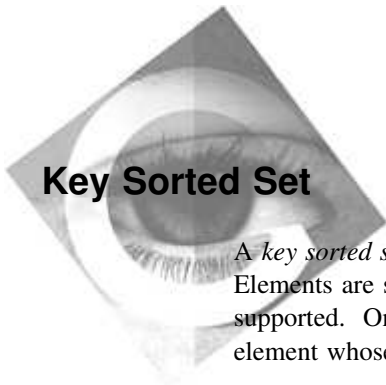
There are 1 words with 2 letters.

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

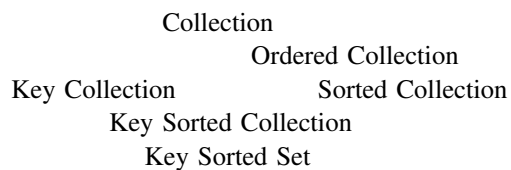
A *key sorted set* is an ordered collection of zero or more elements that have a key. Elements are sorted according to the value of their key field. Element equality is not supported. Only elements with unique keys are supported. A request to add an element whose key already exists is ignored.



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.

 The figure “Combination of Flat Collection Properties” 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



Variants and Header Files

`IKeySortedSet`, the first class in the table below, is the default implementation variant.

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 `ivkssset.h` header file instead of the header file that you would normally use without Visual Builder.

| Class Name | Header File | Implementation Variant |
|--------------------------------------|------------------------|------------------------|
| <code>IKeySortedSet</code> | <code>ikss.h</code> | AVL tree |
| <code>IGKeySortedSet</code> | <code>ikss.h</code> | AVL tree |
| <code>IKeySortedSetAsAvlTree</code> | <code>ikssavl.h</code> | AVL tree |
| <code>IGKeySortedSetAsAvlTree</code> | <code>ikssavl.h</code> | AVL tree |


Key Sorted Set

| Class Name | Header File | Implementation Variant |
|--------------------------|-------------|------------------------|
| IKeySortedSetAsBstTree | ikssbst.h | B* tree |
| IGKeySortedSetAsBstTree | ikssbst.h | B* tree |
| IKeySortedSetAsList | iksslst.h | List |
| IGKeySortedSetAsList | iksslst.h | List |
| IKeySortedSetAsTable | iksstab.h | Table |
| IGKeySortedSetAsTable | iksstab.h | Table |
| IKeySortedSetAsDilTable | ikssdil.h | Diluted table |
| IGKeySortedSetAsDilTable | ikssdil.h | Diluted table |

Members

All members of flat collections are described in “Introduction to Flat Collections” on page 96. The following members are provided for key sorted set:

| Method | Page | Method | Page |
|----------------------------|------|---------------------------|------|
| Constructor | 99 | key | 115 |
| Copy Constructor | 99 | lastElement | 116 |
| Destructor | 99 | locateElementWithKey | 116 |
| operator= | 100 | locateNextElementWithKey | 118 |
| add | 101 | locateOrAddElementWithKey | 119 |
| addAllFrom | 102 | maxNumberOfElements | 120 |
| addOrReplaceElementWithKey | 107 | newCursor | 120 |
| allElementsDo | 108 | numberOfElements | 121 |
| anyElement | 109 | positionAt | 121 |
| compare | 109 | removeAll | 123 |
| containsAllKeysFrom | 110 | removeAt | 124 |
| containsElementWithKey | 111 | removeAtPosition | 124 |
| copy | 111 | removeElementWithKey | 125 |
| elementAt | 112 | removeFirst | 125 |
| elementAtPosition | 113 | removeLast | 126 |
| elementWithKey | 113 | replaceAt | 126 |
| firstElement | 114 | replaceElementWithKey | 126 |
| isBounded | 115 | setToFirst | 127 |
| isEmpty | 115 | setToLast | 127 |
| isFirst | 115 | setToNext | 128 |
| isFull | 115 | setToPosition | 129 |
| isLast | 115 | setToPrevious | 129 |

Key Sorted Set also defines a cursor that inherits from `IOrderedCursor`.  The members for `IOrderedCursor` are described in “Cursor” on page 258.

Key Sorted Set

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

Key Sorted Set as AVL Tree

`IKeySortedSetAsAvlTree` *<Element, Key>*
`IGKeySortedSetAsAvlTree` *<Element, Key, KCOps>*

The implementation of the class `IKeySortedSetAsAvlTree` requires the following element and key-type functions:

Element Type

- Copy constructor
- Assignment
- Destructor
- Key access

Key Type

Ordering relation

Key Sorted Set as B* Tree

`IKeySortedSetAsBstTree` *<Element, Key>*
`IKeySortedSetAsBstTree` *<Element, Key, KCOps>*

The implementation of the class `IKeySortedSetAsBstTree` 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 as List

IKeySortedSetAsList *<Element, Key>*
IGKeySortedSetAsList *<Element, Key, KCOps>*

The implementation of the class IKeySortedSetAsList requires the following element and key-type functions:

Element Type

- Copy constructor
- Destructor
- Assignment
- Key access

Key Type

Ordering relation

Key Sorted Set as Table

IKeySortedSetAsTable *<Element, Key>*
IGKeySortedSetAsTable *<Element, Key, KCOps>*

The implementation of the class IKeySortedSetAsTable requires the following element and key-type functions:

Element Type

- Copy constructor
- Destructor
- Assignment
- Key access

Key Type

Ordering relation

Key Sorted Set

Key Sorted Set as Diluted Table

```
IKeySortedSetAsDilTable <Element, Key>  
IGKeySortedSetAsDilTable <Element, Key, KCOps>
```

The implementation of the class `IKeySortedSetAsDilTable` requires the following element and key-type functions:

Element Type


- Copy constructor
- Destructor
- Assignment
- Key access

Key Type

Ordering relation

Abstract Class

```
IAKeySortedSet<Element,Key>
```

For polymorphism, you can use the corresponding abstract class, `IAKeySortedSet`, which is found in the `iakss.h` header file.  See the section on *Polymorphism and the Collections* in the *Open Class Library User's Guide* for further information.

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.

Key Sorted Set

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 674 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 <ikss.h>

// Let's use the default Heap:
#include <iheap.h>

typedef IKeySortedSet<Parcel, IString> 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.elementAtWithKey("27AmTo").arrivedAt(
        "Atlanta", today);
    circulating.elementAtWithKey("25WaSt").arrivedAt(
        "Amsterdam", today);
    circulating.elementAtWithKey("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.elementAtWithKey("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;
```

Key Sorted Set

```
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 KeySorted 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);
forICursor(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);
    forICursor(pcursor) {
        os << pcursor.element() << "\n";
    }
    return os;
}

ostream& operator<<(ostream& os, ParcelHeap const& parcels) {
    ParcelHeap::Cursor pcursor(parcels);
    forICursor(pcursor) {
        os << pcursor.element() << "\n";
    }
    return os;
}

IBoolean 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:

Key Sorted Set

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.

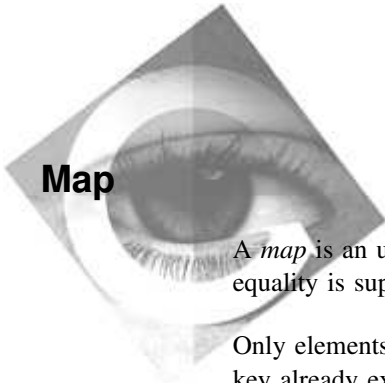
The situation at day 17:
Parcels in circulation:
25DuKa: From Dublin(day 16) to Kairo
 is at Paris since day 17.
25WaSt: From Washington(day 15) to Stockholm
 is at Amsterdam since day 17.
26LoAt: From London(day 8) to Athens
 is at London since day 8.
27AmTo: From Amsterdam(day 10) to Toronto
 is at 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.




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.

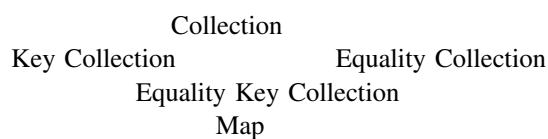


An example of using a map is a program that translates integer values between the ranges of 0 and 20 to their written equivalents, or between written numbers and their numeric values. Two maps are created, one with the integer values as keys, one with the written equivalents as keys. You can enter a 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.

 The figure “Behavior of add for Unique and Multiple Collections” 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.

 The figure “Combination of Flat Collection Properties” 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



Variants and Header Files

IMap, the first class in the table below, is the default implementation variant.

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 |
| IMapAsAvlTree | imapavl.h | AVL Tree |
| IGMapAsAvlTree | imapavl.h | AVL Tree |
| IMapAsBstTree | imapbst.h | B* tree |
| IGMapAsBstTree | imapbst.h | B* tree |
| IMapAsList | imaplst.h | List |
| IGMapAsList | imaplst.h | List |
| IMapAsTable | imaptab.h | Table |
| IGMapAsTable | imaptab.h | Table |
| IMapAsDilTable | imapdil.h | Diluted table |
| IGMapAsDilTable | imapdil.h | Diluted table |
| IMapAsHshTable | imaphsh.h | Hash table |
| IGMapAsHshTable | imaphsh.h | Hash table |


Members

All members of flat collections are described in “Introduction to Flat Collections” on page 96. The following members are provided for map:

| Method | Page | Method | Page |
|----------------------------|------|---------------------------|------|
| Constructor | 99 | containsAllKeysFrom | 110 |
| Copy Constructor | 99 | containsElementWithKey | 111 |
| Destructor | 99 | copy | 111 |
| operator!= | 100 | differenceWith | 112 |
| operator= | 100 | elementAt | 112 |
| operator== | 100 | elementWithKey | 113 |
| add | 101 | intersectionWith | 114 |
| addAllFrom | 102 | isBounded | 115 |
| addDifference | 105 | isEmpty | 115 |
| addIntersection | 106 | isFull | 115 |
| addOrReplaceElementWithKey | 107 | key | 115 |
| addUnion | 107 | locate | 116 |
| allElementsDo | 108 | locateElementWithKey | 116 |
| anyElement | 109 | locateOrAdd | 118 |
| contains | 110 | locateOrAddElementWithKey | 119 |
| containsAllFrom | 110 | maxNumberOfElements | 120 |

Map

| Method | Page | Method | Page |
|----------------------|------|-----------------------|------|
| newCursor | 120 | replaceAt | 126 |
| numberOfElements | 121 | replaceElementWithKey | 126 |
| remove | 122 | setToFirst | 127 |
| removeAll | 123 | setToNext | 128 |
| removeAt | 124 | unionWith | 130 |
| removeElementWithKey | 125 | | |

Map also defines a cursor that inherits from `IElementCursor`.  The members for `IElementCursor` are described in “Cursor” on page 258.

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 as AVL Tree

`IMapAsAvlTree` *<Element, Key>*
`IGMapAsAvlTree` *<Element, Key, EKCOps>*

The implementation of the class `IMapAsAvlTree` requires the following element and key-type functions:

Element Type

- Copy constructor
- Destructor
- Assignment

Map

- Equality test
- Key access

Key Type

Ordering relation

Map as B* Tree

IMapAsBstTree <Element, Key>
IGMapAsBstTree <Element, Key, EKCOps>

The implementation of the class IMapAsBstTree 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 as List

IMapAsList <Element, Key>
IGMapAsList <Element, Key, EKCOps>

The implementation of the class IMapAsSortedList requires the following element and key-type functions:

Element Type

- Copy constructor
- Destructor
- Assignment
- Equality test
- Key access

Key Type

Ordering relation

Map

Map as Table

IMapAsTable <Element, Key>
IGMapAsTable <Element, Key, EKCOps>

The implementation of the class IMapAsTable requires the following element and key-type functions:

Element Type

- Copy constructor
- Destructor
- Assignment
- Equality test
- Key access

Key Type

Ordering relation

Map as Diluted Table

IMapAsDilTable <Element, Key>
IGMapAsDilTable <Element, Key, EKCOps>

The implementation of the class IMapAsDilTable requires the following element and key-type functions:

Element Type

- Copy constructor
- Destructor
- Assignment
- Equality test
- Key access

Key Type

Ordering relation

Map as Hash Table

IMapAsHshTable <Element, Key>
IGMapAsHshTable <Element, Key, EKEHOps>

The implementation of the class IMapAsHshTable requires the following element and key-type functions:

Element Type

Map


- Copy constructor
- Destructor
- Assignment
- Equality test
- Key access

Key Type

- Equality test
- Hash function

Abstract Class

`IAMap<Element,Key>`


For polymorphism, you can use the corresponding abstract class, `IAMap`, which is found in the `iamap.h` header file.  See the section on *Polymorphism and the Collections* in the *Open Class Library User's Guide* for further information.

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 674 for the code of the `transelem.h` file.



```
// transtab.C - An example of using a Map
#include "transelem.h"

// Get the standard operation classes:
#include <istdops.h>

#include "trmapops.h"

// char const translationTable[256] = ....
#include "xebc2asc.h"
```

Map

```
/*-----*\
| 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 IMapOnHashKeySet
    < TranslationElement, char, TranslationOpsE2A > TransE2AMap;

typedef IMapOnHashKeySet
    < 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;
}
```

Map

```
#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:

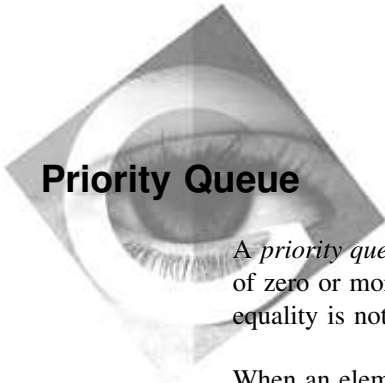
Hex: 24 37 20 20 28 3d 44 6f 6c 6c 61 72 20 73 65 76 65 6e 29

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 the figure “Abstract Class Hierarchy” 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.

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 | ipqu.h | List |
| IGPriorityQueue | ipqu.h | List |
| IPriorityQueueAsList | ipqulst.h | List |
| IGPriorityQueueAsList | ipqulst.h | List |
| IPriorityQueueAsTable | ipqutab.h | Table |
| IGPriorityQueueAsTable | ipqutab.h | Table |
| IPriorityQueueAsDilTable | ipqudil.h | Diluted table |
| IGPriorityQueueAsDilTable | ipqudil.h | Diluted table |

Members

All members of flat collections are described in “Introduction to Flat Collections” on page 96. The following members are provided for priority queue:

| Method | Page | Method | Page |
|------------------------|------|---------------------------|------|
| Constructor | 99 | isFull | 115 |
| Copy Constructor | 99 | isLast | 115 |
| Destructor | 99 | key | 115 |
| operator= | 100 | lastElement | 116 |
| add | 101 | locateElementWithKey | 116 |
| addAllFrom | 102 | locateNextElementWithKey | 118 |
| allElementsDo | 108 | locateOrAddElementWithKey | 119 |
| anyElement | 109 | maxNumberOfElements | 120 |
| compare | 109 | newCursor | 120 |
| containsAllKeysFrom | 110 | numberOfDifferentKeys | 121 |
| containsElementWithKey | 111 | numberOfElements | 121 |
| copy | 111 | numberOfElementsWithKey | 121 |
| dequeue | 112 | positionAt | 121 |
| elementAt | 112 | removeAll | 123 |
| elementAtPosition | 113 | removeFirst | 125 |
| elementWithKey | 113 | setToFirst | 127 |
| enqueue | 113 | setToLast | 127 |
| firstElement | 114 | setToNext | 128 |
| isBounded | 115 | setToNextWithDifferentKey | 128 |
| isEmpty | 115 | setToPosition | 129 |
| isFirst | 115 | setToPrevious | 129 |

Priority queue also defines a cursor that inherits from `IOrderedCursor`.  The members for `IOrderedCursor` are described in “Cursor” on page 258.

Priority Queue

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 as List

IPriorityQueueAsList <Element, Key>
IGPriorityQueueAsList <Element, Key, KCOps>

The implementation of the class IPriorityQueueAsList requires the following element and key-type functions:

Element Type

- Copy constructor
- Destructor
- Assignment
- Key access

Key Type

Ordering relation

Priority Queue as Table

IPriorityQueueAsTable <Element, Key>
IGPriorityQueueAsTable <Element, Key, KCOps>

The implementation of the class IPriorityQueueAsTable requires the following element and key-type functions:

Priority Queue

Element Type

- Copy constructor
- Destructor
- Assignment
- Key access

Key Type

Ordering relation

Priority Queue as Diluted Table

`IPriorityQueueAsDilTable <Element, Key>`
`IGPriorityQueueAsDilTable <Element, Key, KCops>`

The implementation of the class `IPriorityQueueAsDilTable` requires the following element and key-type functions:

Element Type


- Copy constructor
- Destructor
- Assignment
- Key access

Key Type

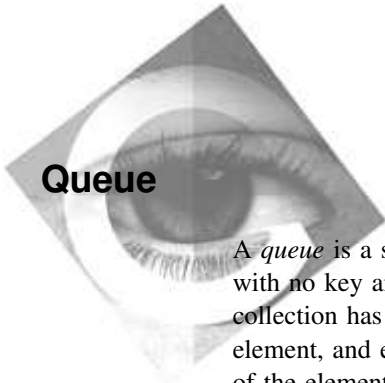
Ordering relation

Abstract Class

`IAPriorityQueue<Element,Key>`

For polymorphism, you can use the corresponding abstract class, `IAPriorityQueue`, which is found in the `iapqu.h` header file.  See the section on *Polymorphism and the Collections* in the *Open Class Library User's Guide* for further information.

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.

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 | iqu.h | List |
| IGQueue | iqu.h | List |
| IQueueAsList | iqulst.h | List |
| IGQueueAsList | iqulst.h | List |
| IQueueAsTable | iqustab.h | Table |
| IGQueueAsTable | iqustab.h | Table |
| IQueueAsDilTable | iqudil.h | Diluted table |
| IGQueueAsDilTable | iqudil.h | Diluted table |

Members

All members of flat collections are described in “Introduction to Flat Collections” on page 96. The following members are provided for queue:

| Method | Page | Method | Page |
|-------------------|------|---------------------|------|
| Constructor | 99 | isBounded | 115 |
| Copy Constructor | 99 | isEmpty | 115 |
| Destructor | 99 | isFirst | 115 |
| operator= | 100 | isFull | 115 |
| add | 101 | isLast | 115 |
| addAllFrom | 102 | lastElement | 116 |
| addAsLast | 103 | maxNumberOfElements | 120 |
| allElementsDo | 108 | newCursor | 120 |
| anyElement | 109 | numberOfElements | 121 |
| compare | 109 | positionAt | 121 |
| copy | 111 | removeAll | 123 |
| dequeue | 112 | removeFirst | 125 |
| elementAt | 112 | setToFirst | 127 |
| elementAtPosition | 113 | setToLast | 127 |
| enqueue | 113 | setToNext | 128 |
| firstElement | 114 | setToPosition | 129 |

Queue also defines a cursor that inherits from `IOrderedCursor`.  The members for `IOrderedCursor` are described in “Cursor” on page 258.

Queue

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 as List

IQueueAsList <Element>
IGQueueAsList <Element, StdOps>

The implementation of the class IQueueAsList requires the following element functions:

Element Type

- Copy constructor
- Destructor
- Assignment

Queue as Table

IQueueAsTable <Element>
IGQueueAsTable <Element, StdOps>

The implementation of the class IDequeAsTable requires the following element functions:

Element Type

- Copy constructor
- Destructor
- Assignment

Queue as Diluted Table

IQueueAsDilTable <Element>
IGQueueAsDilTable <Element, StdOps>

Queue


The implementation of the class `IQueueAsDilTable` requires the following element functions:

Element Type

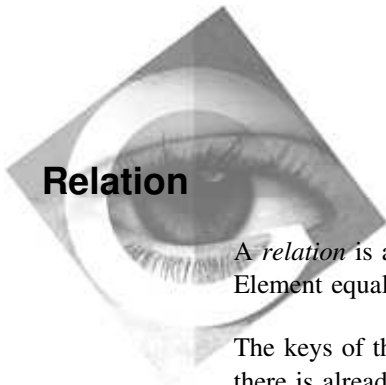
- Copy constructor
- Destructor
- Assignment

Abstract Class

`IAQueue<Element>`

For polymorphism, you can use the corresponding abstract class, `IAQueue`, which is found in the `iaqu.h` header file.  See the section on *Polymorphism and the Collections* in the *Open Class Library User's Guide* for further information.


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.

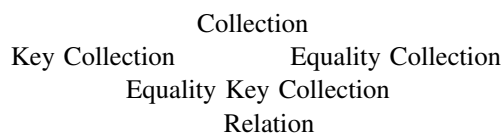
 The figure “Behavior of add for Unique and Multiple Collections” in the *Open Class Library User's Guide* illustrates the differences in behavior between map, relation, key set, and key bag when identical elements and elements with the same key are added.



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.

 The figure “Combination of Flat Collection Properties” 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



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`. If you want to use polymorphism, you can replace these class implementation variants by the reference class.

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 96. The following members are provided for relation:

| Method | Page | Method | Page |
|----------------------------|------|---------------------------|------|
| Constructor | 99 | key | 115 |
| Copy Constructor | 99 | locate | 116 |
| Destructor | 99 | locateElementWithKey | 116 |
| operator!= | 100 | locateNextElementWithKey | 118 |
| operator= | 100 | locateOrAdd | 118 |
| operator== | 100 | locateOrAddElementWithKey | 119 |
| add | 101 | maxNumberOfElements | 120 |
| addAllFrom | 102 | newCursor | 120 |
| addDifference | 105 | numberOfDifferentKeys | 121 |
| addIntersection | 106 | numberOfElements | 121 |
| addOrReplaceElementWithKey | 107 | numberOfElementsWithKey | 121 |
| addUnion | 107 | remove | 122 |
| allElementsDo | 108 | removeAll | 123 |
| anyElement | 109 | removeAllElementsWithKey | 123 |
| contains | 110 | removeAt | 124 |
| containsAllFrom | 110 | removeElementWithKey | 125 |
| containsAllKeysFrom | 110 | replaceAt | 126 |
| containsElementWithKey | 111 | replaceElementWithKey | 126 |
| copy | 111 | setToFirst | 127 |
| differenceWith | 112 | setToNext | 128 |
| elementAt | 112 | setToNextWithDifferentKey | 128 |
| elementWithKey | 113 | unionWith | 130 |
| intersectionWith | 114 | | |
| isBounded | 115 | | |
| isEmpty | 115 | | |
| isFull | 115 | | |

Relation also defines a cursor that inherits from IElementCursor. The members for IElementCursor are described in “Cursor” on page 258.

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

Relation


- Assignment
- Key access
- Equality test

Key Type

- Equality test
- Hash function

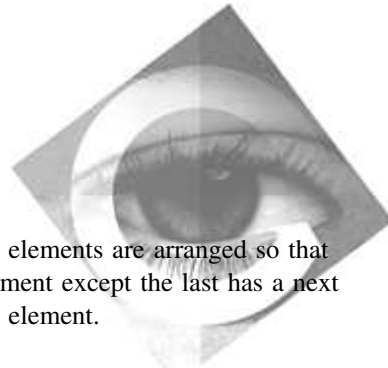
Abstract Class

`IARelation<Element,Key>`


For polymorphism, you can use the corresponding abstract class, `IARelation`, which is found in the `iare1.h` header file.  See the section on *Polymorphism and the Collections* in the *Open Class Library User's Guide* for further information.

The required functions are the same as the required functions of the concrete base class.

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 142 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.

 The figure “Combination of Flat Collection Properties” in the *Open Class Library User's Guide* shows the properties of a sequence and its relationship to other flat collections.

Derivation

Collection
 Ordered Collection
 Sequential Collection
 Sequence

Variants and Header Files

ISequance, the first class in the table below, is the default implementation variant.

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 | List |
| IGSequence | iseq.h | List |
| ISequenceAsList | iseqlst.h | List |
| IGSequenceAsList | iseqlst.h | List |
| | | |
| ISequenceAsTable | iseqtab.h | Table |
| IGSequenceAsTable | iseqtab.h | Table |
| | | |
| ISequenceAsDilTable | iseqdil.h | Diluted table |
| IGDilTable | iseqdil.h | Diluted table |

Members

All members of flat collections are described in “Introduction to Flat Collections” on page 96. The following members are provided for sequence:

| Method | Page | Method | Page |
|-------------------|------|---------------------|------|
| Constructor | 99 | isLast | 115 |
| Copy Constructor | 99 | lastElement | 116 |
| Destructor | 99 | maxNumberOfElements | 120 |
| operator= | 100 | newCursor | 120 |
| add | 101 | numberOfElements | 121 |
| addAllFrom | 102 | positionAt | 121 |
| addAsFirst | 103 | removeAll | 123 |
| addAsLast | 103 | removeAt | 124 |
| addAsNext | 104 | removeAtPosition | 124 |
| addAsPrevious | 104 | removeFirst | 125 |
| addAtPosition | 105 | removeLast | 126 |
| allElementsDo | 108 | replaceAt | 126 |
| anyElement | 109 | replaceAt | 126 |
| compare | 109 | reverse | 127 |
| copy | 111 | setToLast | 127 |
| elementAt | 112 | setToNext | 128 |
| elementAtPosition | 113 | setToPosition | 129 |
| firstElement | 114 | setToPrevious | 129 |
| isBounded | 115 | sort | 129 |
| isEmpty | 115 | | |
| isFirst | 115 | | |
| isFull | 115 | | |

Sequence also defines a cursor that inherits from `IOrderedCursor`.  The members for `IOrderedCursor` are described in “Cursor” on page 258.

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

Sequence as List

`ISequenceAsList` *<Element>*
`IGSequenceAsList` *<Element, StdOps>*

The implementation of the class `ISequenceAsList` requires the following element functions:

Element Type

- Copy constructor
- Destructor
- Assignment

Sequence as Table

`ISequenceAsTable` *<Element>*
`IGSequenceAsTable` *<Element, StdOps>*

The implementation of the class `ISequenceAsTable` requires the following element functions:

Element Type

- Copy constructor
- Destructor
- Assignment

Sequence

Sequence as Diluted Table

```
ISequenceAsDilTable <Element>  
IGSequenceAsDilTable <Element, StdOps>
```


The implementation of the class `ISequenceAsDilTable` requires the following element functions:

Element Type

- Default constructor
- Copy constructor
- Destructor
- Assignment

Abstract Class

```
ISequence<Element>
```


For polymorphism, you can use the corresponding abstract class, `ISequence`, which is found in the `iaseq.h` header file.  See the section on *Polymorphism and the Collections* in the *Open Class Library User's Guide* for further information.

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 674 for the code of the `toyword.h` file.

Sequence



```
// 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 IApplicator <Word> WordAppl;

// Test variables to put into the Sequence.

IString wordArray[9] = {
    "the", "quick", "brown", "fox", "jumps",
    "over", "a", "lazy", "dog"
};

// Our Applicator class for use with allElementsDo().

// The alternative method of iteration, using a cursor, does
// not need such an applicator class.
class PrintClass : public WordAppl
{
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 << "Sequence in sorted order:" << endl;
    WL.allElementsDo(Print);
}
```

Sequence

```
// 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:

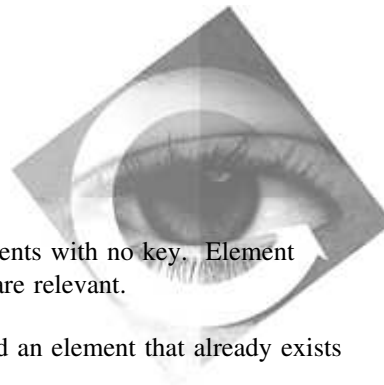
```
a
brown
dog
fox
jumps
lazy
over
quick
the
```

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.

 The figure “Combination of Flat Collection Properties” 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.

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 |
| ISetAsAvlTree | isetavl.h | AVL tree |
| IGSetAsAvlTree | isetavl.h | AVL tree |
| ISetAsBstTree | isetbst.h | B* tree |
| IGSetAsBstTree | isetbst.h | B* tree |
| ISetAsList | isetlst.h | List |
| IGSetAsList | isetlst.h | List |
| ISetAsTable | isettab.h | Table |
| IGSetAsTable | isettab.h | Table |
| ISetAsDilTable | isetdil.h | Diluted table |
| IGSetAsDilTable | isetdil.h | Diluted table |
| ISetAsHshTable | isethsh.h | Hash table |
| IGSetAsHshTable | isethsh.h | Hash table |

Members

All members of flat collections are described in “Introduction to Flat Collections” on page 96. The following members are provided for set:

| Method | Page | Method | Page |
|------------------|------|---------------------|------|
| Constructor | 99 | elementAt | 112 |
| Copy Constructor | 99 | intersectionWith | 114 |
| Destructor | 99 | isBounded | 115 |
| operator!= | 100 | isEmpty | 115 |
| operator= | 100 | isFull | 115 |
| operator== | 100 | locate | 116 |
| add | 101 | locateOrAdd | 118 |
| addAllFrom | 102 | maxNumberOfElements | 120 |
| addDifference | 105 | newCursor | 120 |
| addIntersection | 106 | numberOfElements | 121 |
| addUnion | 107 | remove | 122 |
| allElementsDo | 108 | removeAll | 123 |
| anyElement | 109 | removeAt | 124 |
| contains | 110 | replaceAt | 126 |
| containsAllFrom | 110 | setToFirst | 127 |
| copy | 111 | setToNext | 128 |
| differenceWith | 112 | unionWith | 130 |

Set also defines a cursor that inherits from `IElementCursor`.  The members for `IElementCursor` are described in “Cursor” on page 258.

Template Arguments and Required Functions

Set

```
ISet <Element>
IGSet <Element, COps>
```

The default implementation of the class `ISet` requires the following element functions:

Element Type

- Copy constructor
- Destructor
- Assignment
- Ordering relation

Set as AVL Tree

```
ISetAsAvlTree <Element>
IGSetAsAvlTree <Element, COps>
```

The implementation of the class `ISetAsAvlTree` requires the following element functions:

Element Type

- Copy constructor
- Destructor
- Assignment
- Ordering relation

Set as B* Tree

```
ISetAsBstTree <Element>
IGSetAsBstTree <Element, COps>
```

The implementation of the class `ISetAsBstTree` requires the following element functions:

Element Type

- Copy constructor
- Destructor
- Assignment
- Ordering relation

Set

Set as List

```
ISetAsList <Element>  
IGSetAsList <Element, COps>
```

The implementation of the class ISetAsList requires the following element functions:

Element Type

- Copy constructor
- Destructor
- Assignment
- Ordering relation

Set as Table

```
ISetAsSortedTable <Element>  
IGSetAsTable <Element, COps>
```

The implementation of the class ISetAsTable requires the following element functions:

Element Type

- Copy constructor
- Destructor
- Assignment
- Ordering relation

Set as Diluted Table

```
ISetAsDilTable <Element>  
IGSetAsDilTable <Element, COps>
```

The implementation of the class ISetAsDilTable requires the following element functions:

Element Type

- Copy constructor
- Destructor
- Assignment
- Ordering relation

Set as Hash Table

```
ISetAsHshTable <Element>  
IGSetAsHshTable <Element, EHOps>
```


The implementation of the class ISetAsHshTable requires the following element functions:

Element Type

- Copy constructor
- Destructor
- Assignment
- Equality test
- Hash function

Abstract Class

IASet<Element>

For polymorphism, you can use the corresponding abstract class, IASet, which is found in the `iaset.h` header file.  See the section on *Polymorphism and the Collections* in the *Open Class Library User's Guide* further information.

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 printing the contents of the collection, use an object
// of an applicator class
class PrintClass : public IApplicator<int> {
public:
    virtual IBoolean applyTo(int& i)
    { cout << " " << i << " "; return true;}
};

// Local prototype for the function to display an IntSet.
void List(char *, IntSet &);
```

Set

```
// 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.

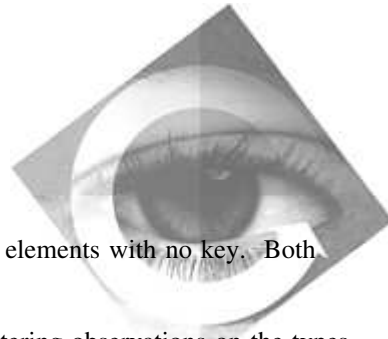
void List(char *Message, IntSet &anIntSet) {
    PrintClass Print;

    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



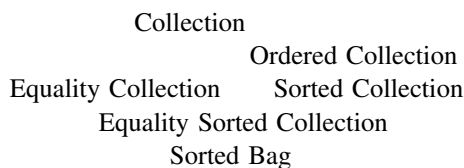
A *sorted bag* is an ordered collection of zero or more elements with no key. Both element equality and multiple elements are supported.



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.

 The figure “Combination of Flat Collection Properties” 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



Variants and Header Files

ISortedBag, the first class in the table below, is the default implementation variant.

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 | isb.h | AVL tree |
| IGSortedBag | isb.h | AVL tree |
| ISortedBagAsAvlTree | isbavl.h | AVL tree |
| IGSortedBagAsAvlTree | isbavl.h | AVL tree |
| ISortedBagAsBstTree | isbbst.h | B* tree |
| IGSortedBagAsBstTree | isbbst.h | B* tree |
| ISortedBagAsList | isblst.h | List |
| IGSortedBagAsList | isblst.h | List |


Sorted Bag

| Class Name | Header File | Implementation Variant |
|-----------------------|-------------|------------------------|
| ISortedBagAsTable | isbt.h | Table |
| IGSortedBagAsTable | isbt.h | Table |
| ISortedBagAsDilTable | isbdil.h | Diluted table |
| IGSortedBagAsDilTable | isbdil.h | Diluted table |

Members

All members of flat collections are described in “Introduction to Flat Collections” on page 96. The following members are provided for sorted bag:

| Method | Page | Method | Page |
|-------------------|------|---------------------------|------|
| Constructor | 99 | isLast | 115 |
| Copy Constructor | 99 | lastElement | 116 |
| Destructor | 99 | locate | 116 |
| operator!= | 100 | locateNext | 117 |
| operator= | 100 | locateOrAdd | 118 |
| operator== | 100 | maxNumberOfElements | 120 |
| add | 101 | newCursor | 120 |
| addAllFrom | 102 | numberOfDifferentElements | 120 |
| addDifference | 105 | numberOfElements | 121 |
| addIntersection | 106 | numberOfOccurrences | 121 |
| addUnion | 107 | positionAt | 121 |
| allElementsDo | 108 | remove | 122 |
| anyElement | 109 | removeAll | 123 |
| compare | 109 | removeAllOccurrences | 124 |
| contains | 110 | removeAt | 124 |
| containsAllFrom | 110 | removeAtPosition | 124 |
| copy | 111 | removeFirst | 125 |
| differenceWith | 112 | removeLast | 126 |
| elementAt | 112 | replaceAt | 126 |
| elementAtPosition | 113 | setToFirst | 127 |
| firstElement | 114 | setToLast | 127 |
| intersectionWith | 114 | setToNext | 128 |
| isBounded | 115 | setToNextDifferentElement | 128 |
| isEmpty | 115 | setToPosition | 129 |
| isFirst | 115 | setToPrevious | 129 |
| isFull | 115 | unionWith | 130 |

Sorted Bag also defines a cursor that inherits from `IOrderedCursor`.  The members for `IOrderedCursor` are described in “Cursor” on page 258.

Template Arguments and Required Functions

Sorted Bag

ISortedBag <Element>
IGSortedBag <Element, COps>

The default implementation of the class ISortedBag requires the following element functions:

Element Type

- Copy constructor
- Destructor
- Assignment
- Ordering relation

Sorted Bag as AVL Tree

ISortedBagAsAvlTree <Element>
IGSortedBagAsAvlTree <Element, COps>

The implementation of the class ISortedBagAsAvlTree requires the following element functions:

Element Type

- Copy constructor
- Destructor
- Assignment
- Ordering relation

Sorted Bag as B* Tree

ISortedBagAsBstTree <Element>
IGSortedBagAsBstTree <Element, COps>

The implementation of the class ISortedBagAsBstTree requires the following element functions:

Element Type

- Copy constructor
- Destructor
- Assignment
- Ordering relation

Sorted Bag

Sorted Bag as List

```
ISortedBagAsList <Element>  
IGSortedBagAsList <Element, COps>
```

The implementation of the class `ISortedBagAsList` requires the following element functions:

Element Type

- Constructor
- Assignment
- Ordering relation

Sorted Bag as Table

```
ISortedBagAsTable <Element>  
IGSortedBagAsTable <Element, COps>
```

The implementation of the class `ISortedBagAsTable` requires the following element functions:

Element Type

- Copy constructor
- Destructor
- Assignment
- Ordering relation

Sorted Bag as Diluted Table

```
ISortedBagAsDilTable <Element>  
IGSortedBagAsDilTable <Element, COps>
```


The implementation of the class `ISortedBagAsDilTable` requires the following element functions:

Element Type

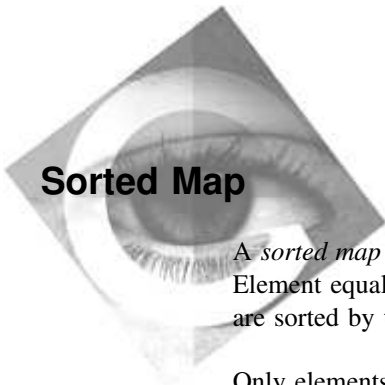
- Copy constructor
- Destructor
- Assignment
- Ordering relation

Abstract Class

IASortedBag<Element>

For polymorphism, you can use the corresponding abstract class, IASortedBag, which is found in the `iasb.h` header file.  See the section on *Polymorphism and the Collections* in the *Open Class Library User's Guide* for further information.

The required functions are the same as the required functions of the concrete base class.




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.

 The figure “Combination of Flat Collection Properties” 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 the figure “Abstract Class Hierarchy” 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.

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 | ism.h | AVL tree |
| IGSortedMap | ism.h | AVL tree |
| ISortedMapAsAvlTree | ismavl.h | AVL tree |
| IGSortedMapAsAvlTree | ismavl.h | AVL tree |
| ISortedMapAsBstTree | ismbst.h | B* tree |
| IGSortedMapAsBstTree | ismbst.h | B* tree |

Sorted Map


| Class Name | Header File | Implementation Variant |
|-----------------------|-------------|------------------------|
| ISortedMapAsList | ismlst.h | List |
| IGSortedMapAsList | ismlst.h | List |
| ISortedMapAsTable | ismtab.h | Table |
| IGSortedMapAsTable | ismtab.h | Table |
| ISortedMapAsDilTable | ismdil.h | Diluted table |
| IGSortedMapAsDilTable | ismdil.h | Diluted table |

Members

All members of flat collections are described in “Introduction to Flat Collections” on page 96. The following members are provided for sorted maps:

| Method | Page | Method | Page |
|----------------------------|------|---------------------------|------|
| Constructor | 99 | isFull | 115 |
| Copy Constructor | 99 | isLast | 115 |
| Destructor | 99 | key | 115 |
| operator!= | 100 | lastElement | 116 |
| operator= | 100 | locate | 116 |
| operator== | 100 | locateElementWithKey | 116 |
| add | 101 | locateNext | 117 |
| addAllFrom | 102 | locateNextElementWithKey | 118 |
| addDifference | 105 | locateOrAdd | 118 |
| addIntersection | 106 | locateOrAddElementWithKey | 119 |
| addOrReplaceElementWithKey | 107 | maxNumberOfElements | 120 |
| addUnion | 107 | newCursor | 120 |
| allElementsDo | 108 | numberOfElements | 121 |
| anyElement | 109 | positionAt | 121 |
| compare | 109 | remove | 122 |
| contains | 110 | removeAll | 123 |
| containsAllFrom | 110 | removeAt | 124 |
| containsAllKeysFrom | 110 | removeAtPosition | 124 |
| containsElementWithKey | 111 | removeElementWithKey | 125 |
| copy | 111 | removeFirst | 125 |
| differenceWith | 112 | removeLast | 126 |
| elementAt | 112 | replaceAt | 126 |
| elementAtPosition | 113 | replaceElementWithKey | 126 |
| elementWithKey | 113 | setToFirst | 127 |
| firstElement | 114 | setToLast | 127 |
| intersectionWith | 114 | setToNext | 128 |
| isBounded | 115 | setToPosition | 129 |
| isEmpty | 115 | setToPrevious | 129 |
| isFirst | 115 | unionWith | 130 |

Sorted Map

Sorted map also defines a cursor that inherits from `IOrderedCursor`.  The members for `IOrderedCursor` are described in “Cursor” on page 258.

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 as AVL Tree

```
ISortedMapAsAvlTree <Element, Key>  
IGSortedMapAsAvlTree <Element, Key, EKCOps>
```

The implementation of the class `ISortedMapAsAvlTree` requires the following element and key-type functions:

Element Type

- Copy constructor
- Destructor
- Assignment
- Key access
- Equality test

Key Type

Ordering relation

Sorted Map as B* Tree

ISortedMapAsBstTree <Element, Key>
IGSortedMapAsBstTree <Element, Key, EKCOps>

The implementation of the class ISortedMapAsBstTree 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 as List

ISortedMapAsList <Element, Key>
IGSortedMapAsList <Element, Key, EKCOps>

The implementation of the class ISortedMapAsList requires the following element and key-type functions:

Element Type

- Copy constructor
- Destructor
- Assignment
- Key access
- Equality test

Key Type

Ordering relation

Sorted Map as Table

ISortedMapAsTable <Element, Key>
IGSortedMapAsTable <Element, Key, EKCOps>

The implementation of the class ISortedMapAsTable requires the following element and key-type functions:

Sorted Map

Element Type

- Copy constructor
- Destructor
- Assignment
- Key access
- Equality test

Key Type

Ordering relation

Sorted Map as Diluted Table

```
ISortedMapAsDilTable <Element, Key>  
IGSortedMapAsDilTable <Element, Key, EKCOps>
```

The implementation of the class `ISortedMapAsDilTable` requires the following element and key-type functions:

Element Type


- Copy constructor
- Destructor
- Assignment
- Key access
- Equality test

Key Type

Ordering relation

Abstract Class

```
IASortedMap<Element, Key>
```

For polymorphism, you can use the corresponding abstract class, `IASortedMap`, which is found in the `iasm.h` header file.  See the section on *Polymorphism and the Collections* in the *Open Class Library User's Guide* for further information.


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 `forICursor` macro and the `allElementsDo()` function.

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). The program uses an input file, `dsu.dat`, rather than call an operating system function to get disk usage information. The input file was created using the `du` command on an AIX system. The input file is contained in the same directory as the sample program.

 See Appendix A, “Header Files for Collection Class Library Coding Examples” on page 674 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);

    ISortedRelation <DiskSpaceUR, int, DSURBySpaceOps>
                    dsurBySpace;
    ISortedRelation <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())
```

Sorted Map

```
        errorExit(39, "Error during read of", fspec);

    cout << "\n\nAll Disk Space Usage records "
        << "sorted (ascending) by name:\n" << endl;

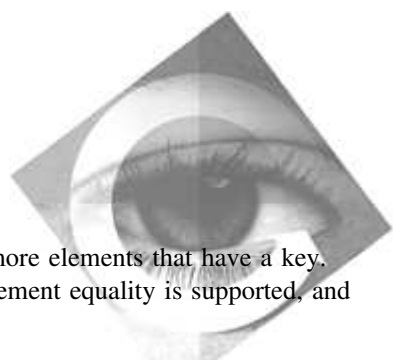
    forICursor(curByName)
        cout << " " << dsurByName.elementAt(curByName) << endl;

    cout << "\n\nAll Disk Space Usage records "
        << "sorted (descending) by space:\n" << endl;

    forICursor(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

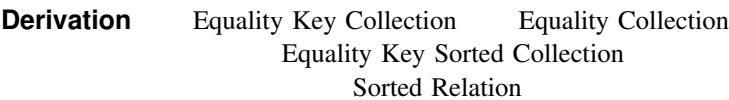
A *sorted relation* is an ordered collection of zero or more elements that have a key. The elements are sorted by the value of their key. Element equality is supported, and the values of the elements are relevant.

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.

The figure “Combination of Flat Collection Properties” 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.



The diagram does not show all bases of sorted relation. See the figure “Abstract Class Hierarchy” 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. |
| | 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 | isr.h | List |
| IGSortedRelation | isr.h | List |
| ISortedRelationAsList | isrslst.h | List |
| IGSortedRelationAsList | isrslst.h | List |
| ISortedRelationAsTable | isrtab.h | Table |
| IGSortedRelationAsTable | isrtab.h | Table |
| ISortedRelationAsDilTable | isrdil.h | Diluted table |
| IGSortedRelationAsDilTable | isrdil.h | Diluted table |


Members

All members of flat collections are described in “Introduction to Flat Collections” on page 96. The following members are provided for sorted relation:

| Method | Page | Method | Page |
|----------------------------|------|---------------------------|------|
| Constructor | 99 | isFull | 115 |
| Copy Constructor | 99 | isLast | 115 |
| Destructor | 99 | key | 115 |
| operator!= | 100 | lastElement | 116 |
| operator= | 100 | locate | 116 |
| operator== | 100 | locateElementWithKey | 116 |
| add | 101 | locateNext | 117 |
| addAllFrom | 102 | locateNextElementWithKey | 118 |
| addDifference | 105 | locateOrAdd | 118 |
| addIntersection | 106 | locateOrAddElementWithKey | 119 |
| addOrReplaceElementWithKey | 107 | maxNumberOfElements | 120 |
| addUnion | 107 | newCursor | 120 |
| allElementsDo | 108 | numberOfDifferentKeys | 121 |
| anyElement | 109 | numberOfElements | 121 |
| compare | 109 | numberOfElementsWithKey | 121 |
| contains | 110 | positionAt | 121 |
| containsAllFrom | 110 | remove | 122 |
| containsAllKeysFrom | 110 | removeAll | 123 |
| containsElementWithKey | 111 | removeAllElementsWithKey | 123 |
| copy | 111 | removeAt | 124 |
| differenceWith | 112 | removeAtPosition | 124 |
| elementAt | 112 | removeElementWithKey | 125 |
| elementAtPosition | 113 | removeFirst | 125 |
| elementWithKey | 113 | removeLast | 126 |
| firstElement | 114 | replaceAt | 126 |
| intersectionWith | 114 | replaceElementWithKey | 126 |
| isBounded | 115 | setToFirst | 127 |
| isEmpty | 115 | setToLast | 127 |
| isFirst | 115 | setToNext | 128 |

Sorted Relation

| Method | Page |
|---------------------------|------|
| setToNextWithDifferentKey | 128 |
| setToPosition | 129 |
| setToPrevious | 129 |
| unionWith | 130 |

Sorted relation also defines a cursor that inherits from `IOrderedCursor`.  The members for `IOrderedCursor` are described in “Cursor” on page 258.

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 as List

`ISortedRelationAsList` *<Element, Key>*
`IGSortedRelationAsList` *<Element, Key, EKCOps>*

The implementation of the class `ISortedRelationAsKey` requires the following element and key-type functions:

Element Type

- Copy constructor
- Destructor
- Assignment
- Key access
- Equality test

Sorted Relation

Key Type

Ordering relation

Sorted Relation as Table

ISortedRelationAsTable <Element, Key>

IGSortedRelationAsTable <Element, Key, EKCOps>

The implementation of the class ISortedRelationAsTable requires the following element and key-type functions:

Element Type

- Copy constructor
- Destructor
- Assignment
- Key access
- Equality test

Key Type

Ordering relation

Sorted Relation as Diluted Table

ISortedRelationAsDilTable <Element, Key>

IGSortedRelationAsDilTable <Element, Key, EKCOps>

The implementation of the class ISortedRelationAsDilTable requires the following element and key-type functions:

Element Type

- Copy constructor
- Destructor
- Assignment
- Key access
- Equality test


Key Type

Ordering relation

Abstract Class


IASortedRelation<Element,Key>

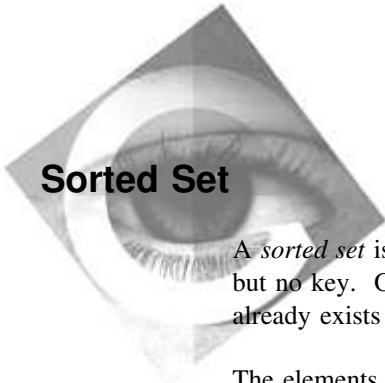
Sorted Relation

For polymorphism, you can use the corresponding abstract class, `IASortedRelation`, which is found in the `iasr.h` header file.  See the section on *Polymorphism and the Collections* in the *Open Class Library User's Guide* for further information.

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 217 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.

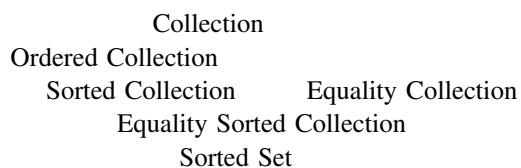
The element with the smallest value currently in a sorted set is called the *first* element. The element with the largest value is called the *last* element. When an element is added, it is placed in the sorted set according to the defined ordering relation.



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.

 The figure “Combination of Flat Collection Properties” 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



Variants and Header Files

ISortedSet, the first class in the table below, is the default implementation variant.

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.

Sorted Set

| Class Name | Header File | Implementation Variant |
|-----------------------|-------------|------------------------|
| ISortedSet | iss.h | AVL tree |
| IGSortedSet | iss.h | AVL tree |
| ISortedSetAsAvlTree | issavl.h | AVL tree |
| IGSortedSetAsAvlTree | issavl.h | AVL tree |
| ISortedSetAsBstTree | issbst.h | B* tree |
| IGSortedSetAsBstTree | issbst.h | B* tree |
| ISortedSetAsList | isslst.h | List |
| IGSortedSetAsList | isslst.h | List |
| ISortedSetAsTable | isstab.h | Table |
| IGSortedSetAsTable | isstab.h | Table |
| ISortedSetAsDilTable | issdil.h | Diluted table |
| IGSortedSetAsDilTable | issdil.h | Diluted table |


Members

All members of flat collections are described in “Introduction to Flat Collections” on page 96. The following members are provided for sorted sets:

| Method | Page | Method | Page |
|-------------------|------|---------------------|------|
| Constructor | 99 | isEmpty | 115 |
| Copy Constructor | 99 | isFirst | 115 |
| Destructor | 99 | isFull | 115 |
| operator!= | 100 | isLast | 115 |
| operator= | 100 | lastElement | 116 |
| operator== | 100 | locate | 116 |
| add | 101 | locateNext | 117 |
| addAllFrom | 102 | locateOrAdd | 118 |
| addDifference | 105 | maxNumberOfElements | 120 |
| addIntersection | 106 | newCursor | 120 |
| addUnion | 107 | positionAt | 121 |
| allElementsDo | 108 | remove | 122 |
| anyElement | 109 | removeAll | 123 |
| compare | 109 | removeAt | 124 |
| contains | 110 | removeAtPosition | 124 |
| containsAllFrom | 110 | removeFirst | 125 |
| copy | 111 | removeLast | 126 |
| differenceWith | 112 | replaceAt | 126 |
| elementAt | 112 | setToFirst | 127 |
| elementAtPosition | 113 | setToLast | 127 |
| firstElement | 114 | setToNext | 128 |
| intersectionWith | 114 | setToPosition | 129 |
| isBounded | 115 | setToPrevious | 129 |

Sorted Set

| Method | Page |
|-----------|------|
| unionWith | 130 |

Sorted Set also defines a cursor that inherits from `IOrderedCursor`.  The members for `IOrderedCursor` are described in “Cursor” on page 258.

Template Arguments and Required Functions

Sorted Set

```
ISortedSet <Element>  
IGSortedSet <Element, COps>
```

The default implementation of the class `ISortedSet` requires the following element functions:

Element Type

- Copy constructor
- Destructor
- Assignment
- Ordering relation

Sorted Set as AVL Tree

```
ISortedSetAsAvlTree <Element>  
IGSortedSetAsAvlTree <Element, EOps>
```

The implementation of the class `ISortedSetAsAvlTree` requires the following element functions:

Element Type

- Copy constructor
- Destructor
- Assignment
- Ordering relation

Sorted Set as B* Tree

```
ISortedSetAsBstTree <Element>  
IGSortedSetAsBstTree <Element, COps>
```


Sorted Set

The default implementation of the class `ISortedSetAsBstTree` requires the following element functions:

Element Type

- Copy constructor
- Destructor
- Assignment
- Ordering relation

Sorted Set as List

`ISortedSetAsList` *<Element>*
`IGSortedSetAsList` *<Element, COps>*

The implementation of the class `ISortedSetAsList` requires the following element functions:

Element Type

- Copy constructor
- Assignment
- Destructor
- Ordering relation

Sorted Set as Table

`ISortedSetAsTable` *<Element>*
`IGSortedSetAsTable` *<Element, COps>*

The implementation of the class `ISortedSetAsTable` requires the following element functions:

Element Type

- Copy constructor
- Destructor
- Assignment
- Ordering relation

Sorted Set as Diluted Table

`ISortedSetAsDilTable` *<Element>*
`IGSortedSetAsDilTable` *<Element, COps>*

Sorted Set


The implementation of the class `ISortedSetAsDilTable` requires the following element functions:

Element Type

- Copy constructor
- Destructor
- Assignment
- Ordering relation

Abstract Class

`ISortedSet<Element>`


For polymorphism, you can use the corresponding abstract class, `ISortedSet`, which is found in the `iass.h` header file.  See the section on *Polymorphism and the Collections* in the *Open Class Library User's Guide* for further information.

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 `forICursor` 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 674 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));

    forICursor(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;
}
```

Sorted Set

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.

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 | istk.h | List |
| IGStack | istk.h | List |
| IStackAsList | istklst.h | List |
| IGStackAsList | istklst.h | List |
| IStackAsTable | istktab.h | Table |
| IGStackAsTable | istktab.h | Table |
| IStackAsDilTable | istkdil.h | Diluted table |
| IGStackAsDilTable | istkdil.h | Diluted table |

Members

All members of flat collections are described in “Introduction to Flat Collections” on page 96. The following members are provided for stack:

| Method | Page | Method | Page |
|-------------------|------|---------------------|------|
| Constructor | 99 | isFull | 115 |
| Copy Constructor | 99 | isLast | 115 |
| Destructor | 99 | lastElement | 116 |
| operator= | 100 | maxNumberOfElements | 120 |
| add | 101 | newCursor | 120 |
| addAllFrom | 102 | numberOfElements | 121 |
| addAsLast | 103 | pop | 121 |
| allElementsDo | 108 | positionAt | 121 |
| anyElement | 109 | push | 122 |
| compare | 109 | removeAll | 123 |
| copy | 111 | removeLast | 126 |
| elementAt | 112 | setToFirst | 127 |
| elementAtPosition | 113 | setToLast | 127 |
| firstElement | 114 | setToNext | 128 |
| isBounded | 115 | setToPosition | 129 |
| isEmpty | 115 | setToPrevious | 129 |
| isFirst | 115 | top | 130 |

Stack also defines a cursor that inherits from `IOrderedCursor`.  The members for `IOrderedCursor` are described in “Cursor” on page 258.

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 as List

```

IStackAsList <Element>
IGStackAsList <Element, StdOps>

```

The implementation of the class IStackAsList requires the following element functions:

Element Type

- Copy constructor
- Destructor
- Assignment

Stack as Table

```

IStackAsTable <Element>
IGStackAsTable <Element, StdOps>

```

The implementation of the class IStackAsTable requires the following element functions:

Element Type

- Copy constructor
- Destructor
- Assignment

Stack as Diluted Table

```

IStackAsDilTable <Element>
IGStackAsDilTable <Element, StdOps>

```

The implementation of the class IStackAsDilTable requires the following element functions:


Element Type

Stack

- Copy constructor
- Destructor
- Assignment

Abstract Class

IStack<Element>

For polymorphism, you can use the corresponding abstract class, IStack, which is found in the iastk.h header file.  See the section on *Polymorphism and the Collections* in the *Open Class Library User's Guide* for further information.

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 applicator class, IConstantApplicator 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 applicator class, IConstantApplicator, 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 IConstantApplicator <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 StackApplicator
{
```


Stack

```
public:
    Boolean 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
```

Stack

```
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

| | |
|---|-----|
| Introduction to Trees | 238 |
| Defining the Traversal Order of Tree Elements | 238 |
| Multiway Tree | 240 |
| Template Arguments and Required Functions | 240 |
| Terms Used | 241 |
| Coding Example for Multiway Tree | 241 |
| Tree Functions | 244 |



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 `IMultiwayTreeIterationOrder` in calls to the following member functions:

- `setToFirst`
- `setToLast`
- `setToNext`
- `setToPrevious`
- `allElementsDo, allSubtreeElementsDo`

 These functions are described in “Multiway Tree” on page 240.

The `IMultiwayTreeIterationOrder` 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

Traversal Order

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 summarized 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).

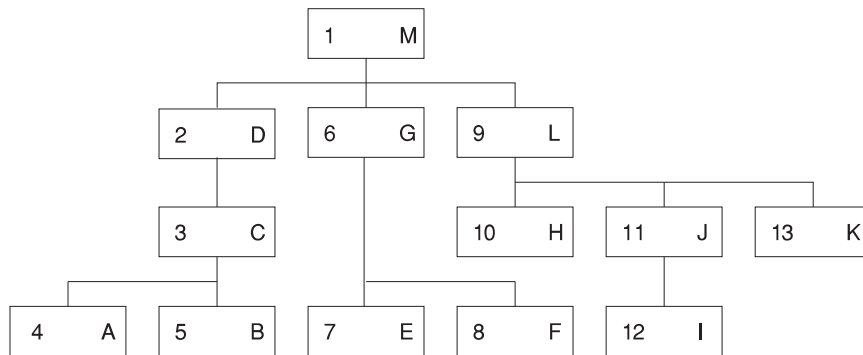
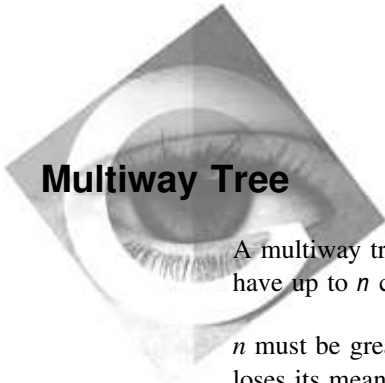


Figure 3. Preorder and Postorder Iteration Methods for Trees



Multiway Tree

A multiway tree, also known as 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 a multiway 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 descendants of a given family member.

Derivation Tree
Multiway Tree

Variants and Header Files IMultiwayTree is the default implementation variant based on tabular tree. IGMultiwayTree is the default implementation variant with generic operations class. Both classes are declared in imwt.h. No reference class exists for tree classes.

Members “Tree Functions” on page 244 lists the member functions for Multiway Tree.

Template Arguments and Required Functions

```
IMultiwayTree <numberOfChildren, Element>  
IGMultiwayTree <numberOfChildren, Element, StdOps>
```

The default implementation of IMultiwayTree 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 <i>given tree</i> . |
| 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 <code>allElementsDo()</code> , <code>allSubtreeElementsDo()</code> , <code>setToNext()</code> , and <code>setToPrevious()</code> . |

Coding Example for Multiway 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 multiway tree
#include <imwt.h>
#include <istring.hpp>
#include <iostream.h>

//////////////////////////////////////
// The tree for this expression is as follows: //
//                                           //
//                               /           //
//                           *         -     //
//                       +       +       7   //
//                   +      8    2    2    4   //
//               ////////////////////////////////////////
```

```
typedef IMultiwayTree<2, IString> BinaryTree;

IBoollean printNode(IString const& node, void* dummy) {
// Prints one node of an multiway tree
cout << node << "|";
return true;
}

void prefixedNotation(BinaryTree const& naryTree) {
// Prints an multiway tree in prefixed notation
naryTree.allElementsDo(printNode , IPreorder);
```

Multiway Tree

```
        cout << endl;
    }

void identifyChildren (IString &child1,
                      IString &child2,
                      BinaryTree &binTree,
                      IMultiwayTreeCursor &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,
                        IMultiwayTreeCursor &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
}
```


Multiway Tree

```
/****** 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);
            }
        }
    }
    while (true);
}
```

Tree Collection Functions

```
        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 multiway trees.

Constructor **IMultiwayTree () ;**

Constructs a tree. The tree is initially empty; that is, it does not contain any nodes.

Copy Constructor **IMultiwayTree (IMultiwayTree <numberOfChildren, Element> const& tree) ;**

Constructs a tree by copying all elements from the given tree.

Exception: IOutOfMemory

Tree Collection Functions

Destructor `~IMultiwayTree () ;`

Removes all elements from this tree.

Side Effects: All cursors of the tree become undefined.

operator= `IMultiwayTree <numberOfChildren, Element>& operator= (`
 `IMultiwayTree <numberOfChildren, Element> 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.
- $(1 \leq position \leq numberOfChildren)$.
- The node denoted by the given cursor (of this tree) must not have a child at the given position.

Exceptions

- `IOutOfMemory`
- `ICursorInvalidException`
- `IPositionInvalidException`
- `ICChildAlreadyExistsException`

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

```
Boolean allElementsDo (
    Boolean (*function) (Element&, void*),
    ITreeIterationOrder iterationOrder,
    void* additionalArgument = 0 ) ;
```

```
Boolean allElementsDo (
    Boolean (*function) (Element const&, void*),
    ITreeIterationOrder iterationOrder,
    void* additionalArgument = 0 ) const;
```

```
Boolean allSubtreeElementsDo ( ITreeCursor const& cursor,
    Boolean (*function) (Element const&, void*),
    ITreeIterationOrder iterationOrder,
    void* additionalArgument = 0 ) const;
```

```
Boolean allSubtreeElementsDo (
    ITreeCursor const& cursor,
    Boolean (*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.

Tree Collection Functions

Preconditions

- The cursor must belong to this tree.
- The cursor must point to an element of this tree.

Exception: `ICursorInvalidException`

allElementsDo, allSubtreeElementsDo

```
IBoolean allElementsDo (
    IApplicator <Element>& applicator,
    ITreeIterationOrder iterationOrder ) ;

IBoolean allElementsDo (
    IConstantApplicator <Element>& applicator,
    ITreeIterationOrder iterationOrder ) const;

IBoolean allSubtreeElementsDo ( ITreeCursor const& cursor,
    IApplicator <Element>& applicator,
    ITreeIterationOrder iterationOrder ) ;

IBoolean allSubtreeElementsDo ( ITreeCursor const& cursor,
    IConstantApplicator <Element>& applicator,
    ITreeIterationOrder iterationOrder ) const;
```

Calls the `applyTo()` function of the given applicator 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`

Tree Collection Functions

attachAsChild, attachSubtreeAsChild

```
void attachAsChild ( ITreeCursor const& cursor,  
    IPosition position,  
    IMultiwayTree <numberOfChildren, Element>& tree ) ;  
  
void attachSubtreeAsChild ( ITreeCursor const& cursor,  
    IPosition position,  
    IMultiwayTree <numberOfChildren, Element>& 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.
- $(1 \leq \textit{position} \leq \textit{numberOfChildren})$.
- 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 (  
    IMultiwayTree <numberOfChildren, Element>& tree ) ;  
  
void attachSubtreeAsRoot (  
    IMultiwayTree <numberOfChildren, Element>& 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`

childPositionAt

```
IPosition childPositionAt (  
    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`

copy, copySubtree

```
void copy (  
    IMultiwayTree <numberOfChildren, Element> const& tree ) ;  
  
void copySubtree (  
    IMultiwayTree <numberOfChildren, Element> 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.

Tree Collection Functions

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.
- $(1 \leq position \leq numberOfChildren)$

Exceptions

- `ICursorInvalidException`
- `IPositionInvalidException`

isEmpty `IBoolean isEmpty () const;`

Returns true if the tree is empty.

Tree Collection Functions

- 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`
- 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.

Tree Collection Functions

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`

removeAll, removeSubtree

`INumber removeAll () ;`

`INumber removeSubtree (ITreeCursor& 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`

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`

Tree Collection Functions

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.
- $(1 \leq position \leq numberOfChildren)$.

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

Tree Collection Functions

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`

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`

Tree Collection Functions

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

Tree Collection Functions

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.

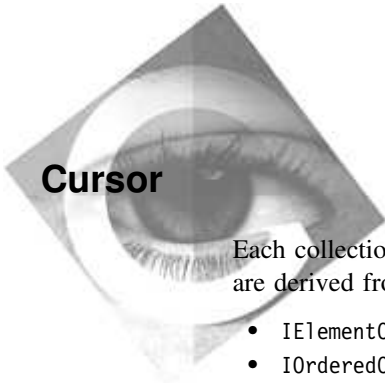
Return Value: Returns true if the tree is not empty.

Exception: ICursorInvalidException

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.

| | |
|---|-----|
| Cursor | 258 |
| Public Member Functions | 259 |
| Tree Cursor | 261 |
| Public Members of Tree Cursor | 261 |
| Applicator and Constant Applicator Classes | 265 |
| Pointer Classes | 266 |
| Members | 266 |
| Coding Example for Managed Element Pointer | 268 |



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 | 259 | <code>operator==</code> | 260 |
| <code>copy</code> | 259 | <code>setToFirst</code> | 260 |
| <code>isValid</code> | 259 | <code>setToLast</code> | 260 |
| <code>invalidate</code> | 259 | <code>setToNext</code> | 260 |
| <code>element</code> | 259 | <code>setToPrevious</code> | 260 |
| <code>operator!=</code> | 259 | | |

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!= (ICursor const& cursor) const;`

Returns true if the cursor does not point to the same element (of the same collection) as the given cursor.

Cursor

operator== `IBoollean operator== (ICursor const& cursor) const;`

Returns true if the cursor points to the same element (of the same collection) as the given cursor.

setToFirst `IBoollean 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 `IBoollean 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 `IBoollean 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`

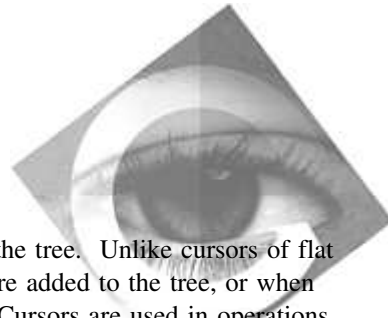
setToPrevious
`IBoollean setToPrevious () ;`

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 `ibtree.h`.

Members Tree Cursor defines the following member functions:

| Method | Page | Method | Page |
|-------------------------|------|---|------|
| Constructor | 261 | <code>setToFirstExistingChild</code> | 262 |
| <code>operator!=</code> | 261 | <code>setToLastExistingChild</code> | 263 |
| <code>operator==</code> | 262 | <code>setToNextExistingChild</code> | 263 |
| <code>element</code> | 262 | <code>setToParent</code> | 263 |
| <code>isValid</code> | 262 | <code>setToPreviousExistingChild</code> | 263 |
| <code>invalidate</code> | 262 | <code>setToRoot</code> | 264 |
| <code>setToChild</code> | 262 | | |

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

- $(1 \leq position \leq numberOfChildren)$.
- 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.

Tree Cursor

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.

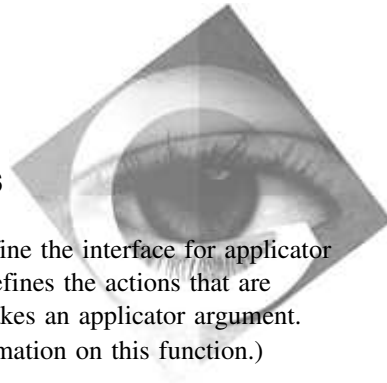
Exception: `ICursorInvalidException`

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`.

Applicator and Constant Applicator Classes



The classes `IApplicator` and `IConstantApplicator` define the interface for applicator objects. The redefinition of the function `applyTo()` defines the actions that are performed with the version of `allElementsDo()` that takes an applicator argument.

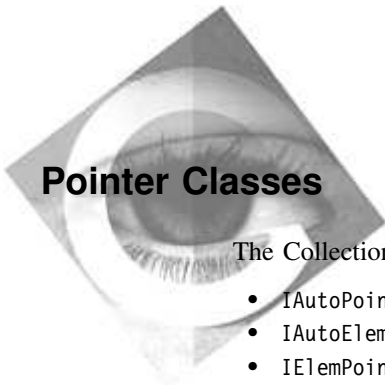
( See “`allElementsDo`” on page 108 for more information on this function.)
Iteration stops when `applyTo()` returns `false`.

( The figure *Iteration Using `allElementsDo`* 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 | <code>iiter.h</code> |
| Members | These classes define only one function, as a virtual function. |
| applyTo | <pre>virtual IBoolean applyTo (Element const& <i>element</i>) = 0;</pre> |

This function applies a series of specified statements or a function to all elements of a collection for which you use the applicator. For example, `myCollection.allElementsDo(myApplicator);` causes the code in the `applyTo()` function that you code for your applicator object `myApplicator` to be applied to all elements of the collection `myCollection`.

For an example on how to use applicators, see “*Iteration Using `allElementsDo`*” on page 97 in the *Open Class Library User's Guide*.




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 Smart Pointers” 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 | 266 | Conversion operator | 267 |
| Copy constructor | 267 | operator-> | 267 |
| Destructors | 267 | operator= | 267 |
| operator* | 267 | operator== | 268 |

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)
```

Pointer Classes

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 - An example of using Collection Class pointers

#include <iostream.h>
#include "graph.h"
#include "line.h"
#include "circle.h"
#include "curve.h"
#include <iptr.h>
#include <ikss.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;
}
```

Pointer Classes

```
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.elementAtKey(70)->draw();
    cPointerToLine->draw();
    pline3->draw();

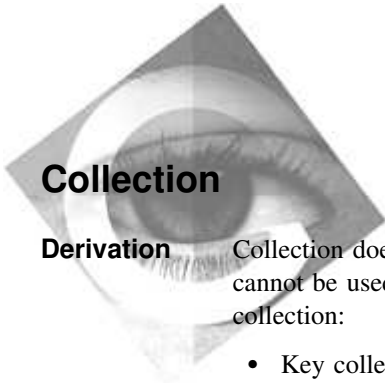
    // Now we are about to end the program. The objects referenced
    // by managed pointers are automatically deleted. See what
    // happens in the output of the program.
    return 0;
}
```

Pointer Classes

Part 6. Abstract Collection Classes

This part describes the abstract Collection Classes.

| | |
|--|-----|
| Collection | 272 |
| Equality Collection | 273 |
| Equality Key Collection | 274 |
| Equality Key Sorted Collection | 275 |
| Equality Sorted Collection | 276 |
| Key Collection | 277 |
| Key Sorted Collection | 278 |
| Ordered Collection | 279 |
| Sequential Collection | 280 |
| Sorted Collection | 281 |




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

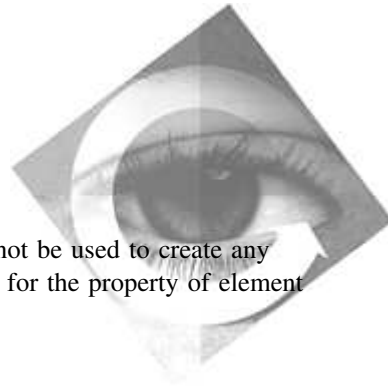
The concrete class heap is defined by collection.

 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 `iac11ct.h`.

Members All the member functions of collection are defined as virtual functions and are described in “Introduction to Flat Collections” on page 96. The following member functions are provided for collection:

| Method | Page | Method | Page |
|-------------------|------|---------------------|------|
| Destructor | 99 | isFull | 115 |
| add | 101 | maxNumberOfElements | 120 |
| addAllFrom | 102 | newCursor | 120 |
| anyElement | 109 | numberOfElements | 121 |
| copy | 109 | removeAll | 123 |
| elementAt | 99 | removeAt | 124 |
| elementAtPosition | 113 | replaceAt | 126 |
| isBounded | 115 | setToFirst | 127 |
| isEmpty | 115 | setToNext | 128 |



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 96, as virtual functions:

| Method | Page | Method | Page |
|-----------------|------|----------------------|------|
| Destructor | 99 | locateOrAdd | 118 |
| contains | 110 | numberOfOccurrences | 121 |
| containsAllFrom | 110 | remove | 122 |
| locate | 116 | removeAllOccurrences | 124 |
| locateNext | 117 | | |

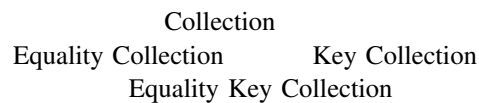


Equality Key Collection

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



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 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 “Abstract Class Hierarchy” in the *Open Class Library User's Guide*.

The following concrete classes are defined by *equality key sorted collection*:

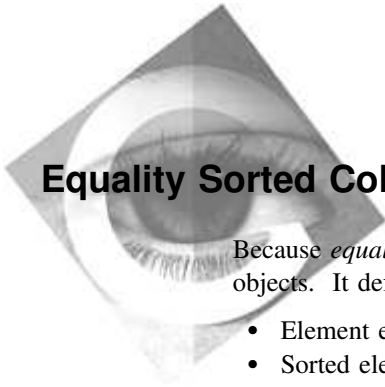
- Sorted map
- Sorted relation

 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

`iaeqsrt.h`.

Members All the members of *equality key sorted collection* are inherited from its base classes.

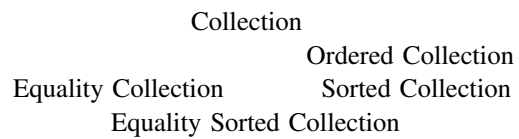


Equality Sorted Collection

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



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

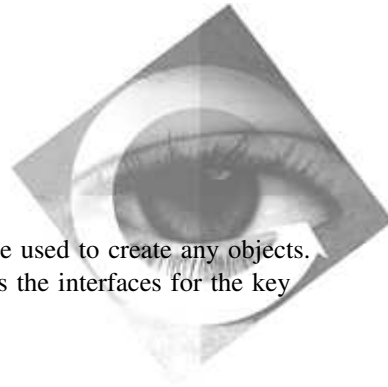
 The figure “The Abstract Class Hierarchy” in the *Open Class Library User's Guide* shows the relationship of equality sorted collection to the class hierarchy.

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:

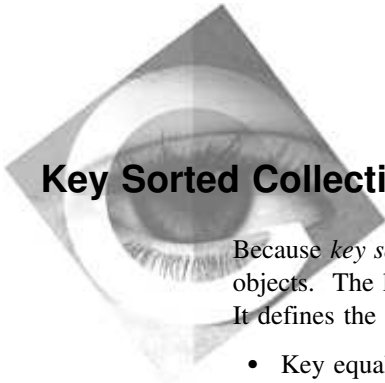
- Key set
- Key bag

 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 96 , as virtual functions:

| Method | Page | Method | Page |
|----------------------------|------|---------------------------|------|
| Destructor | 99 | locateOrAddElementWithKey | 119 |
| addOrReplaceElementWithKey | 107 | numberOfDifferentKeys | 121 |
| containsAllKeysFrom | 110 | numberOfElementsWithKey | 121 |
| containsElementWithKey | 111 | removeAllElementsWithKey | 123 |
| elementWithKey | 113 | removeElementWithKey | 125 |
| key | 115 | replaceElementWithKey | 126 |
| locateElementWithKey | 116 | setToNextWithDifferentKey | 128 |
| locateNextElementWithKey | 118 | | |

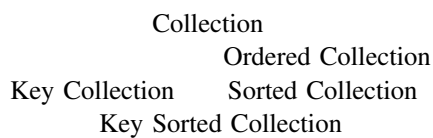


Key Sorted Collection

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



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.




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:

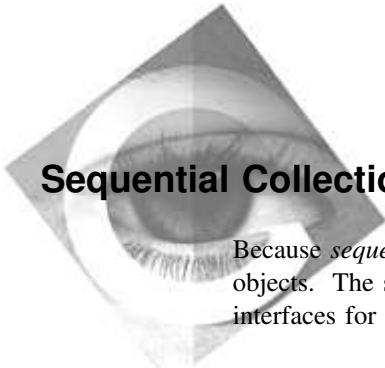
- Sorted collection
- Sequential collection

 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 96, as pure virtual functions:

| Method | Page | Method | Page |
|-------------------|------|------------------|------|
| Destructor | 99 | removeAtPosition | 124 |
| elementAtPosition | 113 | removeFirst | 125 |
| firstElement | 114 | removeLast | 126 |
| isFirst | 115 | setToLast | 127 |
| isLast | 115 | setToPosition | 129 |
| lastElement | 116 | setToPrevious | 129 |
| position | 121 | | |




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 `iasqnt1.h`.

Members Sequential collection defines the following member functions as pure virtual functions:

| Method | Page | Method | Page |
|-------------------|------|---------------------|------|
| Destructor | 99 | isFull | 115 |
| operator= | 100 | isLast | 115 |
| add | 101 | lastElement | 116 |
| addAllFrom | 102 | maxNumberOfElements | 120 |
| addAsFirst | 103 | newCursor | 120 |
| addAsLast | 103 | position | 121 |
| addAsNext | 104 | removeAll | 123 |
| addAsPrevious | 104 | removeAt | 124 |
| addAtPosition | 105 | removeAtPosition | 124 |
| allElementsDo | 108 | removeFirst | 125 |
| anyElement | 109 | removeLast | 126 |
| compare | 109 | replaceAt | 126 |
| elementAt | 112 | setToFirst | 127 |
| elementAtPosition | 113 | setToLast | 127 |
| firstElement | 114 | setToNext | 128 |
| isBounded | 115 | setToPosition | 129 |
| isEmpty | 115 | setToPrevious | 129 |
| isFirst | 115 | sort | 129 |




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, Stream, and Exception Classes

Provide support for the exceptions, trace output, messages, streams, strings, window geometry, and notifications used by the applications you develop.

| | |
|--------------------------------------|-----|
| Class Hierarchy | 286 |
| I0String | 287 |
| IAccessError | 304 |
| IAssertionFailure | 306 |
| IBase | 308 |
| IBase::Version | 313 |
| IBaseErrorInfo | 314 |
| IBaseStream | 320 |
| IBitFlag | 336 |
| IBuffer | 340 |
| ICLibErrorInfo | 358 |
| IContext | 362 |
| IDate | 365 |
| IDBCSBuffer | 376 |
| IDeviceError | 392 |
| IException | 394 |
| IException::TraceFn | 404 |
| IExceptionLocation | 406 |
| IFileStream | 408 |

| | |
|--|-----|
| IGUIErrorInfo | 413 |
| IInvalidParameter | 418 |
| IInvalidRequest | 420 |
| IMemoryStream | 422 |
| IMessageText | 426 |
| IMetaType | 429 |
| IMetaTypeInfo | 432 |
| INotificationEvent | 435 |
| INotifier | 439 |
| IObserver | 444 |
| IObserverList | 447 |
| IObserverList::Cursor | 450 |
| IOutOfMemory | 453 |
| IOutOfSystemResource | 455 |
| IOutOfWindowResource | 457 |
| IPair | 459 |
| IPoint | 467 |
| IPointArray | 470 |
| IRange | 474 |
| IRectangle | 477 |
| IRefCounted | 492 |
| IReference | 495 |

| | |
|---|-----|
| IResourceExhausted | 498 |
| ISize | 500 |
| IStandardNotifier | 503 |
| IString | 508 |
| IStringEnum | 561 |
| IStringParser | 562 |
| IStringParser::SkipWords | 571 |
| IStringTest | 573 |
| IStringTestMemberFn | 577 |
| ISystemErrorInfo | 580 |
| ITime | 584 |
| ITrace | 590 |
| IVBase | 598 |
| IXLibErrorInfo | 601 |

Class Hierarchy

Provide support for the exceptions, trace output, messages, streams, strings, window geometry, and notifications used by the applications you develop.

```

IBase
├── IBitFlag
├── IContext
├── IDate
├── INotificationEvent
├── IPair
│   ├── IPoint
│   ├── IRange
│   └── ISize
├── IPointArray
├── IRectangle
├── IReference
├── IString
│   └── IString
├── IStringParser
├── IStringParser::SkipWords
├── ITime
└── IVBase
    ├── IBaseErrorInfo
    │   ├── ICLibErrorInfo
    │   ├── IGUIErrorInfo
    │   ├── ISystemErrorInfo
    │   └── IXLlibErrorInfo
    ├── IBaseStream
    │   ├── IFileStream
    │   └── IMemoryStream
    ├── IBuffer
    │   └── IDBCSBuffer
    ├── INotifier
    │   └── IStandardNotifier
    ├── IObserver
    ├── IObserverList
    ├── IObserverList::Cursor
    ├── IRefCounted
    ├── IStringTest
    │   └── IStringTestMemberFn
    └── ITrace
IBase::Version
IException
├── IAccessError
├── IAssertionFailure
├── IDeviceError
├── IInvalidParameter
├── IInvalidRequest
├── IResourceExhausted
│   ├── IOutOfMemory
│   ├── IOutOfSystemResource
│   └── IOutOfWindowResource
IException::TraceFn
IExceptionLocation
IMessageText
IMetaType
IMetaTypeInfo
IStringEnum
  
```

I0String



Derivation IBase
 IString
 I0String

Inherited By None.

Header File
 i0string.hpp

| Members | Member | Page | Member | Page |
|---------|---------------|------|-------------------|------|
| | Constructor | 288 | lastIndexOf | 297 |
| | adjustArg | 302 | lastIndexOfAnyBut | 298 |
| | adjustResult | 302 | lastIndexOfAnyOf | 299 |
| | change | 290 | notFound | 303 |
| | charType | 294 | occurrencesOf | 299 |
| | indexOf | 296 | operator [] | 294 |
| | indexOfAnyBut | 296 | overlayWith | 293 |
| | indexOfAnyOf | 297 | remove | 293 |
| | indexOfPhrase | 300 | subString | 295 |
| | indexOfWord | 300 | ~I0String | 290 |
| | insert | 292 | | |

The I0String class provides arrays of characters. Objects of the I0String class are functionally equivalent to objects of the class IString (p. 508) 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

I0String

Constructors

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 User Interface Class 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.

I0String

```
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 anInt );
```

Construct a string with the ASCII representation of an integer numeric value.

```
5 I0String( unsigned anUnsigned );
```

I0String

Construct a string with the ASCII representation of an unsigned numeric value.

6 I0String(long aLong);

Construct a string with the ASCII representation of a long numeric value.

7 I0String(unsigned long anUnsignedLong);

Construct a string with the ASCII representation of an unsigned long numeric value.

8 I0String(long long aLongLong);

9 I0String(unsigned long long anUnsignedLongLong);

10 I0String(short aShort);

Construct a string with the ASCII representation of a short numeric value.

11 I0String(unsigned short anUnsignedShort);

Construct a string with the ASCII representation of an unsigned short numeric value.

12 I0String(double aDouble);

Construct a string with the ASCII representation of a double numeric value.

13 I0String(char aChar);

Construct a string with a copy of the character. The string length is set to 1.

14 I0String(unsigned char anUnsignedChar);

Construct a string with a copy of the unsigned character. The string length is set to 1.

15 I0String(signed char aSignedChar);

Construct a string with a copy of the signed character. The string length is set to 1.

16 I0String(const char* pChar);

Construct a string with a copy of the specified ASCIIZ string.

17 I0String(const unsigned char* pUnsignedChar);

I0String

Construct a string with a copy of the specified unsigned ASCIIZ string.

18 `I0String(const signed char* pSignedChar);`

Construct a string with a copy of the specified signed ASCIIZ string.

19 `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*).

20 `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*).

~I0String

```
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 a NULL-terminated string. The library searches for the pattern string within the receiver's data.

I0String

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.

- 1** I0String&
change(const IString& aPattern,
const IString& aReplacement,
unsigned startPos = 0,
unsigned numChanges = (unsigned) UINT_MAX);
- 2** I0String&
change(const IString& aPattern,
const char* pReplacement,
unsigned startPos = 0,
unsigned numChanges = (unsigned) UINT_MAX);
- 3** I0String&
change(const char* pPattern,
const IString& aReplacement,
unsigned startPos = 0,
unsigned numChanges = (unsigned) UINT_MAX);
- 4** I0String&
change(const char* pPattern,
const char* pReplacement,
unsigned startPos = 0,
unsigned numChanges = (unsigned) UINT_MAX);
- 5** static I0String
change(const IString& aString,
const IString& inputString,
const IString& outputString,
unsigned startPos = 0,
unsigned numChanges = (unsigned) UINT_MAX);

I0String

```
6 static I0String
  change( const IString& aString,
          const IString& inputString,
          const char* pOutputString,
          unsigned startPos = 0,
          unsigned numChanges = ( unsigned ) UINT_MAX );
```

```
7 static I0String
  change( const IString& aString,
          const char* pInputString,
          const IString& outputString,
          unsigned startPos = 0,
          unsigned numChanges = ( unsigned ) UINT_MAX );
```

```
8 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.

```
1 static I0String
  insert( const IString& aString,
          const IString& anInsert,
          unsigned index = ( unsigned ) UINT_MAX,
          char padCharacter = ' ' );
```

```
2 I0String&
  insert( const IString& aString,
          unsigned index = ( unsigned ) UINT_MAX,
          char padCharacter = ' ' );
```

```
3 I0String&
  insert( const char* pString,
          unsigned index = ( unsigned ) UINT_MAX,
          char padCharacter = ' ' );
```

I0String

```
4 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*).

```
1 static I0String
  overlayWith( const IString& aString,
               const IString& anOverlay,
               unsigned index = 0,
               char padCharacter = ' ' );
```

```
2 I0String&
  overlayWith( const IString& aString,
               unsigned index = 0,
               char padCharacter = ' ' );
```

```
3 I0String&
  overlayWith( const char* pString,
               unsigned index = 0,
               char padCharacter = ' ' );
```

```
4 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.

```
1 I0String&
  remove( unsigned startPos );
```

I0String

- 2** I0String&
remove(unsigned startPos,
 unsigned numChars);
- 3** static I0String
remove(const IString& aString,
 unsigned startPos);
- 4** 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.

- 1** const char&
operator [] (unsigned index) const;
- 2** char&
operator [] (unsigned index);
- 3** char&
operator [] (signed index);
- 4** const char&
operator [] (signed index) const;

IString

- 5** `char&`
 `operator [] (unsigned long index);`
- 6** `const char&`
 `operator [] (unsigned long index) const;`
- 7** `char&`
 `operator [] (signed long index);`
- 8** `const char&`
 `operator [] (signed long index) const;`

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.

- 1** `IString`
 `subString(unsigned startPos) const;`
- 2** `IString`
 `subString(unsigned startPos,`
 `unsigned len,`
 `char padCharacter = ' ') const;`

I0String

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. 573).

```
1 unsigned
    indexOf( const IString& aString,
             unsigned startPos = 0 ) const;
```

```
2 unsigned
    indexOf( const char* pString,
             unsigned startPos = 0 ) const;
```

```
3 unsigned
    indexOf( char aCharacter,
             unsigned startPos = 0 ) const;
```

```
4 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. 573) object.

```
1 unsigned
    indexOfAnyBut( const IStringTest& aTest,
                  unsigned startPos = 0 ) const;
```

```
2 unsigned
    indexOfAnyBut( const IString& aString,
                  unsigned startPos = 0 ) const;
```

```
3 unsigned
    indexOfAnyBut( const char* pValidChars,
                  unsigned startPos = 0 ) const;
```

```

4 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. 573) object.

```

1 unsigned
    indexOfAnyOf( char searchChar,
                 unsigned startPos = 0 ) const;

```

```

2 unsigned
    indexOfAnyOf( const IString& searchChars,
                 unsigned startPos = 0 ) const;

```

```

3 unsigned
    indexOfAnyOf( const char* pSearchChars,
                 unsigned startPos = 0 ) const;

```

```

4 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 $0 \leq x \leq startPos$ or IString::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.

```

1 unsigned
    lastIndexOf(
        char aCharacter,
        unsigned endPos = ( unsigned ) ( UINT_MAX - 1 ) ) const;

```

IString

```
2 unsigned
  lastIndexOf(
    const IString& aString,
    unsigned endPos = ( unsigned ) ( UINT_MAX - 1 ) ) const;

3 unsigned
  lastIndexOf(
    const char* pString,
    unsigned endPos = ( unsigned ) ( UINT_MAX - 1 ) ) const;

4 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.

```
1 unsigned
  lastIndexOfAnyBut(
    const IString& validChars,
    unsigned endPos = ( unsigned ) ( UINT_MAX - 1 ) ) const;

2 unsigned
  lastIndexOfAnyBut(
    const char* pValidChars,
    unsigned endPos = ( unsigned ) ( UINT_MAX - 1 ) ) const;

3 unsigned
  lastIndexOfAnyBut(
    char validChar,
    unsigned startPos = ( unsigned ) ( UINT_MAX - 1 ) ) const;
```



```

4 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.

```

1 unsigned
    lastIndexOfAnyOf(
        const IString& searchChars,
        unsigned endPos = ( unsigned ) ( UINT_MAX - 1 ) ) const;

```

```

2 unsigned
    lastIndexOfAnyOf(
        const char* pSearchChars,
        unsigned endPos = ( unsigned ) ( UINT_MAX - 1 ) ) const;

```

```

3 unsigned
    lastIndexOfAnyOf(
        char searchChar,
        unsigned startPos = ( unsigned ) ( UINT_MAX - 1 ) ) const;

```

```

4 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. 525).

```

1 unsigned
    occurrencesOf( const IStringTest& aTest,
        unsigned startPos = 0 ) const;

```

IString

2 unsigned
 occurrencesOf(const IString& aString,
 unsigned startPos = 0) const;

3 unsigned
 occurrencesOf(const char* pString,
 unsigned startPos = 0) const;

4 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 | isAlphabetic | operator += |
| asDouble | isAlphanumeric | operator = |
| asInt | isASCII | operator char * |
| asLongLong | isBinaryDigits | operator signed char * |
| asString | isControl | operator unsigned char * |
| asUnsigned | isDBCS | operator [] |

IString

| IString | | |
|------------------------------------|----------------------------|----------------------------|
| asUnsignedLongLong | isDigits | operator ^ |
| b2c | isGraphics | operator ^= |
| b2d | isHexDigits | operator |
| b2x | isInternationalized | operator = |
| c2b | isLike | operator ~ |
| c2d | isLowerCase | overlayWith |
| c2x | isMBCS | remove |
| center | isPrintable | removeWords |
| change | isPunctuation | reverse |
| charType | isSBCS | rightJustify |
| copy | isUpperCase | size |
| d2b | isValidDBCS | space |
| d2c | isValidMBCS | strip |
| d2x | isWhiteSpace | stripBlanks |
| disableInternationalization | lastIndexOf | stripLeading |
| enableInternationalization | lastIndexOfAnyBut | stripLeadingBlanks |
| includes | lastIndexOfAnyOf | stripTrailing |
| includesDBCS | leftJustify | stripTrailingBlanks |
| includesMBCS | length | subString |
| includesSBCS | lengthOfWord | translate |
| indexOf | lineFrom | upperCase |
| indexOfAnyBut | lowerCase | word |
| indexOfAnyOf | numWords | wordIndexOfPhrase |
| indexOfPhrase | occurrencesOf | words |
| indexOfWord | operator & | x2b |
| insert | operator &= | x2c |
| isAbbreviationFor | operator + | x2d |

| IBase | | |
|-------------|--------------------|-----------------------|
| asDebugInfo | messageFile | setMessageFile |
| asString | messageText | version |

I0String

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.

1 static signed
 adjustArg(signed index);

2 static unsigned
 adjustArg(unsigned index);

adjustResult Adjusts a function result from 1- to 0-based.

1 static signed
 adjustResult(signed index);

2 static unsigned
 adjustResult(unsigned index);

Inherited Protected Functions

| IString | | |
|----------------------|-----------------|----------------|
| applyBitOp | indexOfWord | nullBuf |
| buffer | initBuffer | occurrencesOf |
| change | insert | overlayWith |
| data | isAbbrevFor | setBuffer |
| defaultBuffer | isLike | strip |
| findPhrase | lengthOf | translate |

Public Data

Searches

These members are reimplemented to treat the starting position of the search as a 0-based index.

IString

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 | zero |

| IBase | | |
|-------------|---------------|--|
| recoverable | unrecoverable | |



Derivation IException
IAccessError

Inherited By None.

Header File
iexcbase.hpp

| Members | Member | Page |
|---------|-------------|------|
| | Constructor | 304 |
| | name | 305 |

The IAccessError class represents 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. 498).

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

Constructors

You can construct objects of this class.

IAccessError 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. 403) to specify the severity of the error. The default is unrecoverable.
- Using the macros discussed in IException (p. 394). The User Interface Class Library provides these macros to make creating exceptions easier for you.

IAccessError

```
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. These members are 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 | 306 |
| | name | 307 |

The IAssertionFailure class represents 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. 394) describes IASSERT and the other exception-handling macros.

Public Functions

Constructors

You can construct objects of this class.

IAssertionFailure

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. 403) to specify the severity of the error. The default is unrecoverable.
- Using the macro IASSERT (p. 394). The User Interface Class Library provides this macro to make creating exceptions easier for you.

IAssertionFailure

```
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. These members are 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 | IHighEventParameter |
| | IAcceleratorKey | IKey |
| | IAcceleratorTable | ILowEventParameter |
| | IBitFlag | MenuItem |
| | ICnrAllocator | IMMAudioBuffer |
| | ICollectionViewConstants | INotificationEvent |
| | ICommand | IPair |
| | IContext | IPointArray |
| | ICoordinateSystem | IProcedureAddress |
| | ICritSec | IRectangle |
| | IDate | IReference |
| | IDDEActiveServer | IResourceId |
| | IEventData | IString |
| | IEventParameter1 | IStringGenerator |
| | IEventParameter2 | IStringParser |
| | IEventResult | IStringParser::SkipWords |
| | IFileDialog::Settings | ISWP |
| | IFontDialog::Settings | ISWPArray |
| | IFrameExtension | ITime |
| | IGraphicBundle | ITransformMatrix |
| | IHandle | IVBase |
| | IHelpWindow::Settings | |

Header File

ibase.hpp

Members

| Member | Page | Member | Page |
|-------------|------|----------------|------|
| asDebugInfo | 309 | recoverable | 311 |
| asString | 309 | setMessageFile | 310 |
| messageFile | 309 | unrecoverable | 311 |
| messageText | 310 | version | 309 |

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 that the User Interface Class Library declares in isynonym.hpp.

Public Functions

Diagnostics

Use these members to provide diagnostic information.

asDebugInfo Obtains the diagnostic version of an object's contents. Use this to retrieve an IString representing a hex pointer to the object.

```
IString
asDebugInfo() const;
```

asString 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 Returns the name of the message file used to load library exception text.

```
static char*
messageFile();
```



If you previously called setMessageFile (p. 310) 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).



If you previously called setMessageFile (p. 310) with the name of a message file, the file's name is returned. The default file name is ibmcl.cat.

IBase

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 );
```



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
- The DPATH environment setting
- The APPEND environment setting



The message file is a Windows resource DLL regardless of the extension being used. The search order for this resource DLL is as follows:

- The current working directory
- The PATH environment setting
- The APPEND environment setting



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 derived classes:

```
static IException::Severity
    recoverable;
```

unrecoverable

Synonym for `IException::unrecoverable`. Use this when constructing an `IException` object or one of its derived classes:

```
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.

Associated Globals

operator << `ostream& operator <<(ostream& aStream, const IBase& anObject);`

IBase

Permits any library object to be dumped to an ostream, such as

```
cout << anObject;
```

Note: IBase cannot provide any useful information about the object, so all subclasses should override this function.



IBase::Version

Derivation Inherits from none.

Inherited By None.

Header File
ibase.hpp

| Members | Member | Page |
|---------|--------|------|
| | major | 313 |
| | minor | 313 |

The IBase::Version class defines the version specifier, comprised of major and minor version numbers.

Public Data

Version Data

These members encapsulate versioning data for Class Library objects.

major Provides the major version level of the library. It is incremented by 1 for each new release within a version.

```
unsigned short  
major;
```

minor Provides the minor version level of the library. It starts at 0 for each major version level and is incremented by 1 for each corrective service diskette (CSD).

```
unsigned short  
minor;
```



Inherited By ICLibErrorInfo ISystemErrorInfo
 IGUIErrorInfo IXLibErrorInfo
 IMMEErrorInfo

Header File
 iexcept.hpp

| Members | Member | Page | Member | Page |
|---------|-----------------------|------|-----------------|------|
| | Constructor | 317 | text | 317 |
| | errorId | 317 | throwError | 317 |
| | isAvailable | 317 | ~IBaseErrorInfo | 317 |
| | operator const char * | 317 | | |

The IBaseErrorInfo 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. 418) 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. 420) 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 User Interface Class Library-defined exceptions.

IBaseErrorInfo

| | |
|-----------------|--|
| <i>id</i> | The ID of the message to load from the class library message file. |
| <i>name</i> | A value from the enumeration IBaseErrorInfo::ExceptionType (p. 319), indicating the type of exception to create. |
| <i>severity</i> | A value from the enumeration IException::Severity (p. 403), 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 User Interface Class Library-defined exceptions. It is identical to the ITHROWLIBRARYERROR macro, except it has a fourth parameter:

| | |
|-------------|---|
| <i>text</i> | Replacement text for the retrieved message. |
|-------------|---|

ITHROWERROR

This macro can throw any of the User Interface Class Library-defined exceptions.

| | |
|--------------------|---|
| <i>messageId</i> | The ID of the message to load from the message file. |
| <i>name</i> | A value from the enumeration IBaseErrorInfo::ExceptionType (p. 319), indicating the type of exception to create. |
| <i>severity</i> | A value from the enumeration IException::Severity (p. 403), indicating the severity of the exception. |
| <i>messageFile</i> | The name of the message file to load the exception text from. This name should include the file extension, for example, usermsg.msg. |
| <i>errorGroup</i> | 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

IBaseErrorInfo

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 User Interface Class Library-defined exceptions. It is identical to the ITHROWERROR macro, except it has a fourth parameter:

substitutionText

Substitution text for the retrieved message.



IGUIErrorInfo (p. 413), ISystemErrorInfo (p. 580), and ICLibErrorInfo (p. 358) 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.



In prior releases of User Interface Class Library, this class was named IErrorInfo, but Windows also uses the IErrorInfo class name. To assist in migrating your code to the new class name, add the -DIUSE_IERRORINFO keyword to your compiler flag definitions. This adds a typedef to your code so that IErrorInfo is defined as IBaseErrorInfo. Note that you cannot use this solution when your code also includes the Windows IErrorInfo class.



IXLibErrorInfo (p. 601) 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. 413) and ISystemErrorInfo (p. 580) on AIX, but they have the following default messages:

| | |
|-------------------------|-------------------------------------|
| IGUIErrorInfo | GUI exception condition detected |
| ISystemErrorInfo | System exception condition detected |

Public Functions

Constructors

This is a virtual base class so you cannot create objects of this type without deriving from this class.

IBaseErrorInfo

IBaseErrorInfo

```
IBaseErrorInfo();
```

~IBaseErrorInfo

```
virtual  
~IBaseErrorInfo();
```

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 IBaseErrorInfo 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

IBaseErrorInfo

5. Throw the exception

location An IExceptionLocation (p. 406) object containing the following:

- Function name
- File name
- Line number where the function is called

name Use the enumeration ExceptionType (p. 319) to specify the type of the exception. The default is accessError.

severity Use the enumeration IException::Severity (p. 403) 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

Specifies 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.



IBaseStream

Derivation IBase
IVBase
IBaseStream

Inherited By IFileStream
IMemoryStream

Header File ibasstrm.hpp

| Members | Member | Page | Member | Page |
|---------|-----------------------|------|------------------------|------|
| | Constructor | 330 | position | 329 |
| | context | 327 | read | 321 |
| | defaultContext | 328 | seek | 329 |
| | flush | 328 | seekRelative | 329 |
| | handleReadBufferEmpty | 330 | setContext | 328 |
| | handleWriteBufferFull | 330 | setLogicalEndOfStream | 329 |
| | isWriteable | 328 | setPhysicalEndOfStream | 329 |
| | logicalEndOfStream | 328 | write | 322 |
| | physicalEndOfStream | 328 | ~IBaseStream | 320 |

The IBaseStream class is an abstract class that provides operators for reading and writing data structures to and from a stream. A stream is an abstraction that can accept and provide serial binary data. A stream can represent memory, a file, or other concrete objects.

Public Functions

Constructors

You cannot construct objects of this abstract base class.

~IBaseStream

| | | | | |
|-----------------|--|------------|-----------|--------------|
| virtual | | <u>Win</u> | <u>PM</u> | <u>Motif</u> |
| ~IBaseStream(); | | Y | N | N |

Reading and Writing Arrays

Use these members to read and write arrays of the C++ built-in types to a stream.

read Reads a wide variety of data types from a stream.

| | | | | |
|----------|---|-------------------------------|------------------------------|---------------------------------|
| 1 | virtual IBaseStream& read(signed char target [], unsigned long count); | <u>Win</u> <i>Y</i> | <u>PM</u> <i>N</i> | <u>Motif</u> <i>N</i> |
|----------|---|-------------------------------|------------------------------|---------------------------------|

Use this function to read an array of unsigned char values from the stream.

The parameters are the following:

| | |
|---------------|---|
| <i>target</i> | The area to store the unsigned char values that are read. |
| <i>count</i> | The number of unsigned char values to be read. |

| | | | | |
|----------|---|-------------------------------|------------------------------|---------------------------------|
| 2 | virtual IBaseStream& read(unsigned char target [], unsigned long count); | <u>Win</u> <i>Y</i> | <u>PM</u> <i>N</i> | <u>Motif</u> <i>N</i> |
|----------|---|-------------------------------|------------------------------|---------------------------------|

Use this function to read an array of unsigned char values from the stream. The parameters are the following:

| | |
|---------------|---|
| <i>target</i> | The area to store the unsigned char values that are read. |
| <i>count</i> | The number of unsigned char values to be read. |

| | | | | |
|----------|---|-------------------------------|------------------------------|---------------------------------|
| 3 | virtual IBaseStream& read(short target [], unsigned long count); | <u>Win</u> <i>Y</i> | <u>PM</u> <i>N</i> | <u>Motif</u> <i>N</i> |
|----------|---|-------------------------------|------------------------------|---------------------------------|

Use this function to read an array of short values from the stream. The parameters are the following:

| | |
|---------------|---|
| <i>target</i> | The area to store the short values that are read. |
| <i>count</i> | The number of short values to be read. |

| | | | | |
|----------|--|-------------------------------|------------------------------|---------------------------------|
| 4 | virtual IBaseStream& read(unsigned short target [], unsigned long count); | <u>Win</u> <i>Y</i> | <u>PM</u> <i>N</i> | <u>Motif</u> <i>N</i> |
|----------|--|-------------------------------|------------------------------|---------------------------------|

Use this function to read an array of unsigned short values from the stream. The parameters are the following:

| | |
|---------------|--|
| <i>target</i> | The area to store the unsigned short values that are read. |
| <i>count</i> | The number of unsigned short values to be read. |

IBaseStream

| | | | | | | | | |
|-------------------|---|--|-------------------|------------------|---------------------|----------|----------|----------|
| 5 | <code>virtual IBaseStream& read(long target [], unsigned long count);</code> | <table border="0"><tr><td><u>Win</u></td><td><u>PM</u></td><td><u>Motif</u></td></tr><tr><td><i>Y</i></td><td><i>N</i></td><td><i>N</i></td></tr></table> | <u>Win</u> | <u>PM</u> | <u>Motif</u> | <i>Y</i> | <i>N</i> | <i>N</i> |
| <u>Win</u> | <u>PM</u> | <u>Motif</u> | | | | | | |
| <i>Y</i> | <i>N</i> | <i>N</i> | | | | | | |

Use this function to read an array of long values from the stream. The parameters are the following:

| | |
|---------------|--|
| <i>target</i> | The area to store the long values that are read. |
| <i>count</i> | The number of long values to be read. |

| | | | | | | | | |
|-------------------|--|--|-------------------|------------------|---------------------|----------|----------|----------|
| 6 | <code>virtual IBaseStream& read(unsigned long target [], unsigned long count);</code> | <table border="0"><tr><td><u>Win</u></td><td><u>PM</u></td><td><u>Motif</u></td></tr><tr><td><i>Y</i></td><td><i>N</i></td><td><i>N</i></td></tr></table> | <u>Win</u> | <u>PM</u> | <u>Motif</u> | <i>Y</i> | <i>N</i> | <i>N</i> |
| <u>Win</u> | <u>PM</u> | <u>Motif</u> | | | | | | |
| <i>Y</i> | <i>N</i> | <i>N</i> | | | | | | |

Use this function to read an array of unsigned long values from the stream.

The parameters are the following:

| | |
|---------------|---|
| <i>target</i> | The area to store the unsigned long values that are read. |
| <i>count</i> | The number of unsigned long values to be read. |

| | | | | | | | | |
|-------------------|--|--|-------------------|------------------|---------------------|----------|----------|----------|
| 7 | <code>virtual IBaseStream& read(float target [], unsigned long count);</code> | <table border="0"><tr><td><u>Win</u></td><td><u>PM</u></td><td><u>Motif</u></td></tr><tr><td><i>Y</i></td><td><i>N</i></td><td><i>N</i></td></tr></table> | <u>Win</u> | <u>PM</u> | <u>Motif</u> | <i>Y</i> | <i>N</i> | <i>N</i> |
| <u>Win</u> | <u>PM</u> | <u>Motif</u> | | | | | | |
| <i>Y</i> | <i>N</i> | <i>N</i> | | | | | | |

Use this function to read an array of float values from the stream. The parameters are the following:

| | |
|---------------|---|
| <i>target</i> | The area to store the float values that are read. |
| <i>count</i> | The number of float values to be read. |

| | | | | | | | | |
|-------------------|---|--|-------------------|------------------|---------------------|----------|----------|----------|
| 8 | <code>virtual IBaseStream& read(double target [], unsigned long count);</code> | <table border="0"><tr><td><u>Win</u></td><td><u>PM</u></td><td><u>Motif</u></td></tr><tr><td><i>Y</i></td><td><i>N</i></td><td><i>N</i></td></tr></table> | <u>Win</u> | <u>PM</u> | <u>Motif</u> | <i>Y</i> | <i>N</i> | <i>N</i> |
| <u>Win</u> | <u>PM</u> | <u>Motif</u> | | | | | | |
| <i>Y</i> | <i>N</i> | <i>N</i> | | | | | | |

Use this function to read an array of double values from the stream. The parameters are the following:

| | |
|---------------|--|
| <i>target</i> | The area to store the double values that are read. |
| <i>count</i> | The number of double values to be read. |

write Use these member functions to write a wide variety of data types to a stream.

IBaseStream

| | | | | | | | | |
|-------------------|---|--|-------------------|------------------|---------------------|----------|----------|----------|
| 1 | <pre>virtual IBaseStream& write(const long source [], unsigned long count);</pre> | <table border="0"><tr><td><u>Win</u></td><td><u>PM</u></td><td><u>Motif</u></td></tr><tr><td><i>Y</i></td><td><i>N</i></td><td><i>N</i></td></tr></table> | <u>Win</u> | <u>PM</u> | <u>Motif</u> | <i>Y</i> | <i>N</i> | <i>N</i> |
| <u>Win</u> | <u>PM</u> | <u>Motif</u> | | | | | | |
| <i>Y</i> | <i>N</i> | <i>N</i> | | | | | | |

Use this function to write an array of long values to the stream.

The parameters are the following:

| | |
|---------------|---|
| <i>source</i> | The address of the long values to be written. |
| <i>count</i> | The number of long values to be written. |

| | | | | | | | | |
|-------------------|--|--|-------------------|------------------|---------------------|----------|----------|----------|
| 2 | <pre>virtual IBaseStream& write(const signed char source [], unsigned long count);</pre> | <table border="0"><tr><td><u>Win</u></td><td><u>PM</u></td><td><u>Motif</u></td></tr><tr><td><i>Y</i></td><td><i>N</i></td><td><i>N</i></td></tr></table> | <u>Win</u> | <u>PM</u> | <u>Motif</u> | <i>Y</i> | <i>N</i> | <i>N</i> |
| <u>Win</u> | <u>PM</u> | <u>Motif</u> | | | | | | |
| <i>Y</i> | <i>N</i> | <i>N</i> | | | | | | |

Use this function to write an array of signed char values to the stream.

The parameters are the following:

| | |
|---------------|--|
| <i>source</i> | The address of the signed char values to be written. |
| <i>count</i> | The number of signed char values to be written. |

| | | | | | | | | |
|-------------------|--|--|-------------------|------------------|---------------------|----------|----------|----------|
| 3 | <pre>virtual IBaseStream& write(const unsigned char source [], unsigned long count);</pre> | <table border="0"><tr><td><u>Win</u></td><td><u>PM</u></td><td><u>Motif</u></td></tr><tr><td><i>Y</i></td><td><i>N</i></td><td><i>N</i></td></tr></table> | <u>Win</u> | <u>PM</u> | <u>Motif</u> | <i>Y</i> | <i>N</i> | <i>N</i> |
| <u>Win</u> | <u>PM</u> | <u>Motif</u> | | | | | | |
| <i>Y</i> | <i>N</i> | <i>N</i> | | | | | | |

Use this function to write an array of unsigned char values to the stream.

The parameters are the following:

| | |
|---------------|--|
| <i>source</i> | The address of the unsigned char values to be written. |
| <i>count</i> | The number of unsigned char values to be written. |

| | | | | | | | | |
|-------------------|--|--|-------------------|------------------|---------------------|----------|----------|----------|
| 4 | <pre>virtual IBaseStream& write(const short source [], unsigned long count);</pre> | <table border="0"><tr><td><u>Win</u></td><td><u>PM</u></td><td><u>Motif</u></td></tr><tr><td><i>Y</i></td><td><i>N</i></td><td><i>N</i></td></tr></table> | <u>Win</u> | <u>PM</u> | <u>Motif</u> | <i>Y</i> | <i>N</i> | <i>N</i> |
| <u>Win</u> | <u>PM</u> | <u>Motif</u> | | | | | | |
| <i>Y</i> | <i>N</i> | <i>N</i> | | | | | | |

Use this function to write an array of short values to the stream.

The parameters are the following:

| | |
|---------------|--|
| <i>source</i> | The address of the short values to be written. |
| <i>count</i> | The number of short values to be written. |

IBaseStream

| | | | | |
|----------|---|-------------------------------|------------------------------|---------------------------------|
| 5 | <pre>virtual IBaseStream& write(const unsigned short source [], unsigned long count);</pre> | <u>Win</u> <i>Y</i> | <u>PM</u> <i>N</i> | <u>Motif</u> <i>N</i> |
|----------|---|-------------------------------|------------------------------|---------------------------------|

Use this function to write an array of unsigned short values to the stream.

The parameters are the following:

| | |
|---------------|---|
| <i>source</i> | The address of the unsigned short values to be written. |
| <i>count</i> | The number of unsigned short values to be written. |

| | | | | |
|----------|--|-------------------------------|------------------------------|---------------------------------|
| 6 | <pre>virtual IBaseStream& write(const unsigned long source [], unsigned long count);</pre> | <u>Win</u> <i>Y</i> | <u>PM</u> <i>N</i> | <u>Motif</u> <i>N</i> |
|----------|--|-------------------------------|------------------------------|---------------------------------|

Use this function to write an array of unsigned long values to the stream.

The parameters are the following:

| | |
|---------------|--|
| <i>source</i> | The address of the unsigned long values to be written. |
| <i>count</i> | The number of unsigned long values to be written. |

| | | | | |
|----------|--|-------------------------------|------------------------------|---------------------------------|
| 7 | <pre>virtual IBaseStream& write(const float source [], unsigned long count);</pre> | <u>Win</u> <i>Y</i> | <u>PM</u> <i>N</i> | <u>Motif</u> <i>N</i> |
|----------|--|-------------------------------|------------------------------|---------------------------------|

Use this function to write an array of float values to the stream.

The parameters are the following:

| | |
|---------------|--|
| <i>source</i> | The address of the float values to be written. |
| <i>count</i> | The number of float values to be written. |

| | | | | |
|----------|---|-------------------------------|------------------------------|---------------------------------|
| 8 | <pre>virtual IBaseStream& write(const double source [], unsigned long count);</pre> | <u>Win</u> <i>Y</i> | <u>PM</u> <i>N</i> | <u>Motif</u> <i>N</i> |
|----------|---|-------------------------------|------------------------------|---------------------------------|

Use this function to write an array of double values to the stream.

The parameters are the following:

| | |
|---------------|---|
| <i>source</i> | The address of the double values to be written. |
| <i>count</i> | The number of double values to be written. |

Reading and Writing Primitives

Use these members to read and write the C++ built-in types to a stream.

read Reads a wide variety of data types from a stream.

| | | | | |
|----------|--|-------------------------------|------------------------------|---------------------------------|
| 1 | virtual IBaseStream& read(float& target); | <u>Win</u> <i>Y</i> | <u>PM</u> <i>N</i> | <u>Motif</u> <i>N</i> |
|----------|--|-------------------------------|------------------------------|---------------------------------|

Use this function to read a float value from the stream into the *target*.

| | | | | |
|----------|--|-------------------------------|------------------------------|---------------------------------|
| 2 | virtual IBaseStream& read(void* target, size_t byteCount); | <u>Win</u> <i>Y</i> | <u>PM</u> <i>N</i> | <u>Motif</u> <i>N</i> |
|----------|--|-------------------------------|------------------------------|---------------------------------|

Use this function to read untyped data from the stream.

The parameters are the following:

target The area to store the untyped data that is read.
byteCount The number of bytes of data to be read.

| | | | | |
|----------|---|-------------------------------|------------------------------|---------------------------------|
| 3 | virtual IBaseStream& read(char* target); | <u>Win</u> <i>Y</i> | <u>PM</u> <i>N</i> | <u>Motif</u> <i>N</i> |
|----------|---|-------------------------------|------------------------------|---------------------------------|

Use this function to read a null-terminated character string from the stream into the *target*.

| | | | | |
|----------|--|-------------------------------|------------------------------|---------------------------------|
| 4 | virtual IBaseStream& read(signed char& target); | <u>Win</u> <i>Y</i> | <u>PM</u> <i>N</i> | <u>Motif</u> <i>N</i> |
|----------|--|-------------------------------|------------------------------|---------------------------------|

Use this function to read a signed char value from the stream into the *target*.

| | | | | |
|----------|--|-------------------------------|------------------------------|---------------------------------|
| 5 | virtual IBaseStream& read(unsigned char& target); | <u>Win</u> <i>Y</i> | <u>PM</u> <i>N</i> | <u>Motif</u> <i>N</i> |
|----------|--|-------------------------------|------------------------------|---------------------------------|

Use this function to read an unsigned char value from the stream into the *target*.

| | | | | |
|----------|--|-------------------------------|------------------------------|---------------------------------|
| 6 | virtual IBaseStream& read(short& target); | <u>Win</u> <i>Y</i> | <u>PM</u> <i>N</i> | <u>Motif</u> <i>N</i> |
|----------|--|-------------------------------|------------------------------|---------------------------------|

Use this function to read a short value from the stream into the *target*.

| | | | | |
|----------|---|-------------------------------|------------------------------|---------------------------------|
| 7 | virtual IBaseStream& read(unsigned short& target); | <u>Win</u> <i>Y</i> | <u>PM</u> <i>N</i> | <u>Motif</u> <i>N</i> |
|----------|---|-------------------------------|------------------------------|---------------------------------|

IBaseStream

Use this function to read an unsigned short value from the stream into the *target*.

| | | | | |
|----------|---|-----------------|----------------|-------------------|
| 8 | virtual IBaseStream& read(long& target); | <u>Win</u> Y | <u>PM</u> N | <u>Motif</u> N |
|----------|---|-----------------|----------------|-------------------|

Use this function to read a long value from the stream into the *target*.

| | | | | |
|----------|--|-----------------|----------------|-------------------|
| 9 | virtual IBaseStream& read(unsigned long& target); | <u>Win</u> Y | <u>PM</u> N | <u>Motif</u> N |
|----------|--|-----------------|----------------|-------------------|

Use this function to read an unsigned long value from the stream into the *target*.

| | | | | |
|-----------|---|-----------------|----------------|-------------------|
| 10 | virtual IBaseStream& read(double& target); | <u>Win</u> Y | <u>PM</u> N | <u>Motif</u> N |
|-----------|---|-----------------|----------------|-------------------|

Use this function to read a double value from the stream into the *target*.

write

Use these member functions to write a wide variety of data types to a stream.

| | | | | |
|----------|--|-----------------|----------------|-------------------|
| 1 | virtual IBaseStream& write(unsigned char source); | <u>Win</u> Y | <u>PM</u> N | <u>Motif</u> N |
|----------|--|-----------------|----------------|-------------------|

Use this function to write the unsigned char *source* value to the stream.

| | | | | |
|----------|---|-----------------|----------------|-------------------|
| 2 | virtual IBaseStream& write(const void* source, size_t byteCount); | <u>Win</u> Y | <u>PM</u> N | <u>Motif</u> N |
|----------|---|-----------------|----------------|-------------------|

Use this function to write untyped data to the stream.

The parameters are the following:

source The address of the untyped data to be written.
byteCount The number of bytes of untyped data to be written.

| | | | | |
|----------|--|-----------------|----------------|-------------------|
| 3 | virtual IBaseStream& write(const char* source); | <u>Win</u> Y | <u>PM</u> N | <u>Motif</u> N |
|----------|--|-----------------|----------------|-------------------|

Use this function to write the null-terminated character string from the *source* to the stream.

| | | | | |
|----------|--|-----------------|----------------|-------------------|
| 4 | virtual IBaseStream& write(signed char source); | <u>Win</u> Y | <u>PM</u> N | <u>Motif</u> N |
|----------|--|-----------------|----------------|-------------------|

IBaseStream

Use this function to write the signed char *source* value to the stream.

| | | | | |
|----------|--|-----------------|----------------|-------------------|
| 5 | virtual IBaseStream& write(short source); | <u>Win</u> Y | <u>PM</u> N | <u>Motif</u> N |
|----------|--|-----------------|----------------|-------------------|

Use this function to write the short *source* value to the stream.

| | | | | |
|----------|---|-----------------|----------------|-------------------|
| 6 | virtual IBaseStream& write(unsigned short source); | <u>Win</u> Y | <u>PM</u> N | <u>Motif</u> N |
|----------|---|-----------------|----------------|-------------------|

Use this function to write an unsigned short value to the stream.

| | | | | |
|----------|---|-----------------|----------------|-------------------|
| 7 | virtual IBaseStream& write(long source); | <u>Win</u> Y | <u>PM</u> N | <u>Motif</u> N |
|----------|---|-----------------|----------------|-------------------|

Use this function to write the long *source* value to the stream.

| | | | | |
|----------|--|-----------------|----------------|-------------------|
| 8 | virtual IBaseStream& write(unsigned long source); | <u>Win</u> Y | <u>PM</u> N | <u>Motif</u> N |
|----------|--|-----------------|----------------|-------------------|

Use this function to write the unsigned long *source* value to the stream.

| | | | | |
|----------|--|-----------------|----------------|-------------------|
| 9 | virtual IBaseStream& write(float source); | <u>Win</u> Y | <u>PM</u> N | <u>Motif</u> N |
|----------|--|-----------------|----------------|-------------------|

Use this function to write the float *source* value to the stream.

| | | | | |
|-----------|---|-----------------|----------------|-------------------|
| 10 | virtual IBaseStream& write(double source); | <u>Win</u> Y | <u>PM</u> N | <u>Motif</u> N |
|-----------|---|-----------------|----------------|-------------------|

Use this function to write the double *source* value to the stream.

Stream Context

Use these members to control access to an IContext (p. 362). A *context* is a dynamic dictionary used to handle multiple references to the same object in a stream.

context Returns the IContext (p. 362) object for this stream.

| | | | |
|-------------------------------|-----------------|----------------|-------------------|
| IContext* context() const; | <u>Win</u> Y | <u>PM</u> N | <u>Motif</u> N |
|-------------------------------|-----------------|----------------|-------------------|

IBaseStream

defaultContext

Returns the default IContext (p. 362) object for this stream.

| | | | |
|-------------------------|------------|-----------|--------------|
| IContext* | <u>Win</u> | <u>PM</u> | <u>Motif</u> |
| defaultContext() const; | <i>Y</i> | <i>N</i> | <i>N</i> |

setContext Sets the context of this stream. Associate this stream with an IContext object.

| | | | |
|----------------------------------|------------|-----------|--------------|
| virtual IBaseStream& | <u>Win</u> | <u>PM</u> | <u>Motif</u> |
| setContext(IContext* context); | <i>Y</i> | <i>N</i> | <i>N</i> |

Stream Positioning, Flushing, and Write Access

Use these members to control the buffering, write access, and positioning of a stream.

flush Flushes the buffer associated with the stream. This is a virtual function whose behavior is determined by the derived class. If the derived class does not override this function, the default behavior does nothing.

| | | | |
|----------------------|------------|-----------|--------------|
| virtual IBaseStream& | <u>Win</u> | <u>PM</u> | <u>Motif</u> |
| flush(); | <i>Y</i> | <i>N</i> | <i>N</i> |

isWriteable Returns an indication of whether or not the stream can be written to. This is a virtual function whose behavior is determined by the derived class. If the derived class does not override this function, the default behavior always returns true.

| | | | |
|----------------------|------------|-----------|--------------|
| virtual Boolean | <u>Win</u> | <u>PM</u> | <u>Motif</u> |
| isWriteable() const; | <i>Y</i> | <i>N</i> | <i>N</i> |

logicalEndOfStream

Returns the logical end of the stream.

| | | | |
|-----------------------------|------------|-----------|--------------|
| virtual Position | <u>Win</u> | <u>PM</u> | <u>Motif</u> |
| logicalEndOfStream() const; | <i>Y</i> | <i>N</i> | <i>N</i> |

physicalEndOfStream

A pure virtual function that returns the physical end of the stream. The behavior of this operator is determined by the derived class.

| | | | |
|----------------------------------|------------|-----------|--------------|
| virtual Position | <u>Win</u> | <u>PM</u> | <u>Motif</u> |
| physicalEndOfStream() const = 0; | <i>Y</i> | <i>N</i> | <i>N</i> |

IBaseStream

position A pure virtual function that returns the current position in the stream. The behavior of this operator is determined by the derived subclass.

| | | | |
|---|-----------------|----------------|-------------------|
| virtual Position position() const = 0; | <u>Win</u> Y | <u>PM</u> N | <u>Motif</u> N |
|---|-----------------|----------------|-------------------|

seek A pure virtual function that moves to an absolute location in the stream. The behavior of this operator is determined by the derived class.

| | | | |
|--|-----------------|----------------|-------------------|
| virtual IBaseStream& seek(Position position) = 0; | <u>Win</u> Y | <u>PM</u> N | <u>Motif</u> N |
|--|-----------------|----------------|-------------------|

seekRelative Moves to a position in the stream relative to the current position. Call this function directly to move to a new position in the stream.

| | | | |
|---|-----------------|----------------|-------------------|
| virtual IBaseStream& seekRelative(PositionDelta offset); | <u>Win</u> Y | <u>PM</u> N | <u>Motif</u> N |
|---|-----------------|----------------|-------------------|

setLogicalEndOfStream

Sets the logical end of stream to the desired location. This is a virtual function that derived classes can override.

| | | | |
|---|-----------------|----------------|-------------------|
| virtual IBaseStream& setLogicalEndOfStream(Position position); | <u>Win</u> Y | <u>PM</u> N | <u>Motif</u> N |
|---|-----------------|----------------|-------------------|

setPhysicalEndOfStream

A pure virtual function that sets the physical end of stream to the desired location.

| | | | |
|--|-----------------|----------------|-------------------|
| virtual IBaseStream& setPhysicalEndOfStream(Position position) = 0; | <u>Win</u> Y | <u>PM</u> N | <u>Motif</u> N |
|--|-----------------|----------------|-------------------|

Inherited Public Functions

| IVBase | | |
|-------------|----------|--|
| asDebugInfo | asString | |

| IBase | | |
|-------------|-------------|----------------|
| asDebugInfo | messageFile | setMessageFile |
| asString | messageText | version |

IBaseStream

Protected Functions

Buffer Management

Derived classes are responsible for overriding these pure virtual functions to implement the reading and writing of data to a specific type of stream.

handleReadBufferEmpty

Handles reading of data from the stream. This is a pure virtual function that must be overridden by derived classes.

| | | | | | | | |
|---|--|--------------|-----------|--------------|----------|----------|----------|
| <pre>virtual void handleReadBufferEmpty(void* target, size_t byteCount) = 0;</pre> | <table><tr><td><u>Win</u></td><td><u>PM</u></td><td><u>Motif</u></td></tr><tr><td><i>Y</i></td><td><i>N</i></td><td><i>N</i></td></tr></table> | <u>Win</u> | <u>PM</u> | <u>Motif</u> | <i>Y</i> | <i>N</i> | <i>N</i> |
| <u>Win</u> | <u>PM</u> | <u>Motif</u> | | | | | |
| <i>Y</i> | <i>N</i> | <i>N</i> | | | | | |

The parameters are the following:

| | |
|------------------|---|
| <i>target</i> | The address to store the data that is read. |
| <i>byteCount</i> | The number of bytes to be read. |

handleWriteBufferFull

Handles writing of data to the stream. This is a pure virtual function that must be overridden by derived classes.

| | | | | | | | |
|---|--|--------------|-----------|--------------|----------|----------|----------|
| <pre>virtual void handleWriteBufferFull(const void* source, size_t byteCount) = 0;</pre> | <table><tr><td><u>Win</u></td><td><u>PM</u></td><td><u>Motif</u></td></tr><tr><td><i>Y</i></td><td><i>N</i></td><td><i>N</i></td></tr></table> | <u>Win</u> | <u>PM</u> | <u>Motif</u> | <i>Y</i> | <i>N</i> | <i>N</i> |
| <u>Win</u> | <u>PM</u> | <u>Motif</u> | | | | | |
| <i>Y</i> | <i>N</i> | <i>N</i> | | | | | |

The parameters are the following:

| | |
|------------------|------------------------------------|
| <i>source</i> | The address of the data to write. |
| <i>byteCount</i> | The number of bytes to be written. |

Constructors

You cannot construct objects of this abstract base class.

IBaseStream

| | | | | | | | |
|---------------------------|--|--------------|-----------|--------------|----------|----------|----------|
| <pre>IBaseStream();</pre> | <table><tr><td><u>Win</u></td><td><u>PM</u></td><td><u>Motif</u></td></tr><tr><td><i>Y</i></td><td><i>N</i></td><td><i>N</i></td></tr></table> | <u>Win</u> | <u>PM</u> | <u>Motif</u> | <i>Y</i> | <i>N</i> | <i>N</i> |
| <u>Win</u> | <u>PM</u> | <u>Motif</u> | | | | | |
| <i>Y</i> | <i>N</i> | <i>N</i> | | | | | |

The default constructor. Only derived classes can call this function because it is protected.

Inherited Protected Data

| IBase | | |
|-------------|---------------|--|
| recoverable | unrecoverable | |

Nested Type Definitions

TypeLength `TypeLength {
 kCharBytes = 1, kSignedCharBytes = 1, kUnsignedCharBytes = 1,
 kShortBytes = 2, kUnsignedShortBytes = 2, kUnsignedLongBytes = 4,
 kLongBytes = 4, kFloatBytes = 4, kDoubleBytes = 8
 };`

Defines the size of primitive data types in bytes.

Position `typedef unsigned long Position;`

A number to identify a location in the stream. For example, the seek function takes a Position parameter to set the location where the next read/write will take place.

VersionNumber `typedef unsigned short VersionNumber;`

A typedef to be used with the readVersion and writeVersion functions. A version number identifies the version of the data in a stream.

PositionDelta `typedef long PositionDelta;`

A number that identifies the offset location. For example, the seekRelative function takes PositionDelta as a parameter to move the location by an amount indicated by this parameter.

Associated Globals

flatten `void flatten(const AType* userObjectToFlatten, IBaseStream& theStream);`

Determines the real type of the object pointed to by an AType pointer and streams out the object to the input stream. If the object has been flattened to the stream previously, it is saved again. Instead, a reference is made in the stream to refer to the already saved object.

IBaseStream

operator <<= IBaseStream& operator <<=(ICollection < AType >&, IBaseStream& s);

Reads the elements of a collection (which can be a set, sequence, bag, and similar elements.) from the stream. The elements must be streamable. If the elements are value-based, they can be either primitive data types (that is, long, short and double) with corresponding streaming operators. Or, they must be objects of a class that supports streaming operators. If the elements are pointer-based, they should be of the type IElementPointer<AType>, where AType is the type of the actual data object.

operator <<= IBaseStream& operator <<=(char* target, IBaseStream& baseStream);

Reads a null-terminated string from *baseStream* and stores it to the buffer pointed to by *target*.

operator <<= IBaseStream& operator <<=(unsigned short& target, IBaseStream& baseStream);

Reads an unsigned short from *baseStream* and stores it to the target buffer.

operator <<= IBaseStream& operator <<=(unsigned long& target, IBaseStream& baseStream);

Reads an unsigned long from *baseStream* and stores it to the target buffer.

operator <<= IBaseStream& operator <<=(char& target, IBaseStream& baseStream);

Reads a null-terminated string from *baseStream* and stores it to the target buffer.

operator <<= IBaseStream& operator <<=(double& target, IBaseStream& baseStream);

Reads a double from *baseStream* and stores it to the target buffer.

operator <<= IBaseStream& operator <<=(IElemPointer < AType >&, IBaseStream& baseStream);

Resurrects the pointed-to element from the stream, without an explicit call to the ::resurrect global function. This is especially useful when used with Collection classes.

operator <<= IBaseStream& operator <<=(float& target, IBaseStream& baseStream);

Reads a float from *baseStream* and stores it to the target buffer.

operator <<= IBaseStream& operator <<=(unsigned char& target, IBaseStream& baseStream);

IBaseStream

Reads an unsigned null-terminated string from *baseStream* and stores it to the target buffer.

operator <<= IBaseStream& operator <<=(short& target, IBaseStream& baseStream);

Reads a short from *baseStream* and stores it to the target buffer.

operator <<= IBaseStream& operator <<=(signed char& target, IBaseStream& baseStream);

Reads a signed null-terminated string from *baseStream* and stores it to the target buffer.

operator <<= IBaseStream& operator <<=(long& target, IBaseStream& baseStream);

Reads a long from *baseStream* and stores it to the target buffer.

operator <<= IBaseStream& operator <<=(IString& target, IBaseStream& baseStream);

Reads an IString from *baseStream* and stores it to the target buffer.

operator >>= IBaseStream& operator >>=(const IString& source, IBaseStream& baseStream);

Writes an IString to *baseStream* from the address specified in *source*.

operator >>= IBaseStream& operator >>=(short source, IBaseStream& baseStream);

Writes a short to *baseStream* from the address specified in *source*.

operator >>= IBaseStream& operator >>=(float source, IBaseStream& baseStream);

Writes a float to *baseStream* from the address specified in *source*.

operator >>= IBaseStream& operator >>=(const IElemPointer < AType >&, IBaseStream& baseStream);

Flattens the pointed-to element to the stream, without an explicit call to the ::flatten global function. This is especially useful when used with Collection classes.

operator >>= IBaseStream& operator >>=(const char* source, IBaseStream& baseStream);

Writes a null-terminated string to *baseStream* from the address specified in *source*.

IBaseStream

operator >>= IBaseStream& operator >>=(unsigned char source, IBaseStream& baseStream);

Writes an unsigned null-terminated string to *baseStream* from the address specified in *source*.

operator >>= IBaseStream& operator >>=(const IACollection < AType >&, IBaseStream& s);

Writes the elements of a collection (which can be a set, sequence, bag, and similar elements) to the stream. The elements must be streamable. If the elements are value-based, they can be either primitive data types (that is, long, short and double) with corresponding streaming operators. Or, they must be objects of a class that supports streaming operators. If the elements are pointer-based, they should be of the type *IElementPointer*<AType>, where AType is the type of the actual data object.

operator >>= IBaseStream& operator >>=(signed char source, IBaseStream& baseStream);

Writes a signed null-terminated string to *baseStream* from the address specified in *source*.

operator >>= IBaseStream& operator >>=(unsigned long source, IBaseStream& baseStream);

Writes an unsigned long to *baseStream* from the address specified in *source*.

operator >>= IBaseStream& operator >>=(long source, IBaseStream& baseStream);

Writes a long to *baseStream* from the address specified in *source*.

operator >>= IBaseStream& operator >>=(char source, IBaseStream& baseStream);

Writes a null-terminated string to *baseStream* from the address specified in *source*.

operator >>= IBaseStream& operator >>=(double source, IBaseStream& baseStream);

Writes a double to *baseStream* from the address specified in *source*.

operator >>= IBaseStream& operator >>=(unsigned short source, IBaseStream& baseStream);

Writes an unsigned short to *baseStream* from the address specified in *source*.

readVersion IBaseStream::VersionNumber readVersion(IBaseStream& fromWhere);

IBaseStream

Reads from the stream, *fromWhere*, and returns a value which is the version number.

resurrect `void resurrect(AType *& theResult, IBaseStream& theStream);`

Performs the opposite of flatten. It restores an object from the input stream. If the object has been resurrected previously, the output parameter, *theResult*, points to that instance.

writeVersion `void writeVersion(IBaseStream& toWhere, const IBaseStream::VersionNumber version = 0);`

Writes the version number to the stream *toWhere*. If the version number parameter is not given, the value 0 is used.



IBitFlag

Derivation

IBase
IBitFlag

Inherited By

| | |
|------------------------------|------------------------------------|
| I3StateCheckBox::Style | IMenu::Style |
| IAnimatedButton::Style | IMenuBar::Style |
| IBaseComboBox::Style | IMenuDrawItemHandler::DrawFlag |
| IBaseListBox::Style | IMenuItem::Attribute |
| IBaseSpinButton::Style | IMenuItem::Style |
| IBitmapControl::Style | IMessageBox::Style |
| IButton::Style | IMultiCellCanvas::Style |
| ICanvas::Style | IMultiLineEdit::Style |
| ICheckBox::Style | INotebook::PageSettings::Attribute |
| ICircularSlider::Style | INotebook::Style |
| ICombobox::Style | INumericSpinButton::Style |
| IContainerControl::Attribute | IOutlineBox::Style |
| IContainerControl::Style | IProgressIndicator::Style |
| IControl::Style | IPushButton::Style |
| ICustomButton::Style | IRadioButton::Style |
| IDMImage::Style | IScrollBar::Style |
| IDrawingCanvas::Style | ISetCanvas::Style |
| IEntryField::Style | ISlider::Style |
| IFileDialog::Style | ISplitCanvas::Style |
| IFontDialog::Style | IStaticText::Style |
| IFrameWindow::Style | ITextSpinButton::Style |
| IGraphicPushButton::Style | IToolBar::Style |
| IGroupBox::Style | IToolBarButton::Style |
| IHelpWindow::Style | IToolBarContainer::Style |
| IIconControl::Style | IViewPort::Style |
| IKey::KeyModifier | IWindow::Style |
| IListBox::Style | |

Header File

ibitflag.hpp

Members

| Member | Page | Member | Page |
|------------------------|------|-------------|------|
| Constructor | 339 | operator != | 338 |
| asExtendedUnsignedLong | 338 | operator == | 338 |
| asUnsignedLong | 338 | setValue | 339 |

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 is 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 let you do the following:

- 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).

IBitFlag

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 );
```

Constructors

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:

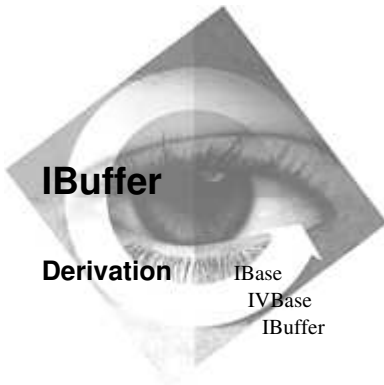
- Existing style objects
- Existing attribute objects
- Combinations of these objects

IBitFlag

```
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 | Member | Page |
|---------------------|------|----------------------|------|
| Constructor | 355 | isMBCS | 345 |
| addRef | 348 | isPrintable | 353 |
| allocate | 354 | isPunctuation | 353 |
| asDebugInfo | 341 | isSBCS | 345 |
| center | 341 | isUpperCase | 353 |
| change | 341 | isValidDBCS | 345 |
| charType | 345 | isValidMBCS | 345 |
| checkAddition | 346 | isWhiteSpace | 353 |
| checkMultiplication | 347 | lastIndexOf | 350 |
| className | 355 | lastIndexOfAnyBut | 350 |
| compare | 341 | lastIndexOfAnyOf | 350 |
| contents | 345 | leftJustify | 342 |
| copy | 342 | length | 346 |
| dbcsTable | 356 | lowerCase | 342 |
| defaultBuffer | 345 | newBuffer | 347 |
| fromContents | 346 | next | 346 |
| includesDBCS | 344 | null | 346 |
| includesMBCS | 344 | operator delete | 354 |
| includesSBCS | 344 | operator new | 354 |
| indexOf | 349 | overflow | 348 |
| indexOfAnyBut | 349 | overlayWith | 342 |
| indexOfAnyOf | 349 | remove | 343 |
| initialize | 355 | removeRef | 348 |
| insert | 342 | reverse | 343 |
| isAlphabetic | 352 | rightJustify | 343 |
| isAlphanumeric | 352 | setDefaultBuffer | 348 |
| isASCII | 352 | startBackwardsSearch | 355 |
| isControl | 352 | startSearch | 356 |
| isDBCS | 344 | strip | 343 |
| isDBCSLead | 356 | subString | 351 |
| isDigits | 352 | translate | 344 |
| isGraphics | 352 | upperCase | 344 |
| isHexDigits | 352 | useCount | 346 |
| isLowerCase | 353 | ~IBuffer | 355 |

The IBuffer class defines the contents of an IString (p. 508).

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 a NULL-terminated string. The library searches for the pattern string within the receiver's data.

sourceLen The length of the source string.

IBuffer

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 uppercase letters in the receiver to lowercase.

```
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*).

IBuffer

```
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 the leading and trailing character or characters. You can specify the character or characters as the following:

- A char* array
- An IStringTest (p. 573) object

The default is white space.

```
1 virtual IBuffer*
    strip( const IStringTest& aTest,
           IStringEnum::StripMode mode );
```

IBuffer

```
2 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 lowercase letters in the receiver to uppercase.

```
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.

IBuffer

```
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.

1

```
const char*  
contents() const;
```

2

```
char*  
contents();
```

defaultBuffer Returns the address of the NULL buffer for the class. This is a static function.

```
static IBuffer*  
defaultBuffer();
```

IBuffer

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 User Interface Class 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.

```
1 virtual char*  
    next( const char* prev );
```

```
2 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 strings' contents 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

```
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. 346) or IBuffer::checkMultiplication (p. 347) detects an overflow.

```
static unsigned
overflow();
```

| Exceptions | |
|----------------|--|
| InvalidRequest | You made an IBuffer request causing an overflow. Typically, this occurs during object construction or during an operation that 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.

- 1** virtual unsigned
 indexOf(const char* pString,
 unsigned len,
 unsigned startPos = 1) const;
- 2** 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. 573) object.

- 1** virtual unsigned
 indexOfAnyBut(const IStringTest& aTest,
 unsigned startPos = 1) const;
- 2** 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. 573) object.

- 1** virtual unsigned
 indexOfAnyOf(const char* pString,
 unsigned len,
 unsigned startPos = 1) const;
- 2** virtual unsigned
 indexOfAnyOf(const IStringTest& aTest,
 unsigned startPos = 1) const;

IBuffer

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 $0 \leq x \leq startPos$. 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.

```
1 virtual unsigned
  lastIndexOf( const char* pString,
               unsigned len,
               unsigned startPos = 0 ) const;
```

```
2 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.

```
1 virtual unsigned
  lastIndexOfAnyBut( const IStringTest& aTest,
                    unsigned startPos = 0 ) const;
```

```
2 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.

IBuffer

If you specify 0 for *startPos*, this function returns 0 indicating the search target was not found.

- 1** virtual unsigned
 lastIndexOfAnyOf(const IStringTest& aTest,
 unsigned startPos = 0) const;
- 2** 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.

- 1** virtual IBuffer*
 subString(unsigned startPos) const;
- 2** virtual IBuffer*
 subString(unsigned startPos,
 unsigned len,
 char padCharacter) const;

Testing

Corresponding IString members use these members to test the buffer's contents.

IBuffer

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 `isctrl()` 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 white-space characters.

White-space 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, white-space characters are those in the range {0x09-0x0D,0x20}.

```
virtual Boolean
    isWhiteSpace() const;
```

Inherited Public Functions

| IVBase | | |
|-------------|----------|--|
| asDebugInfo | asString | |

IBuffer

| 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.

```
1 void  
    operator delete( void* p,  
                    const char* filename,  
                    size_t linenum );
```

```
2 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*.

```
1 void*  
    operator new( size_t t,  
                 const char* filename,  
                 size_t linenum,  
                 unsigned bufLen );
```

```
2 void*  
    operator new( size_t t,  
                 unsigned bufLen );
```


Constructors

You can construct and destruct objects of this class. Constructors require the length of the buffer, which is the value to be stored in the *len* data member.

IBuffer

```
IBuffer( unsigned newLen );
```

Initializes the reference count and terminates the buffer with a NULL.

~IBuffer

```
~IBuffer();
```

Destructor, does nothing.

Implementation

This member helps implement this class.

initialize Initializes the IBuffer object (sets up a NULL buffer, a DBCS table, and so forth).

```
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. 539) using the following criteria:

- 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.

IBuffer

- 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. 525) using the following criteria:

- 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;
```

Testing

Corresponding `IString` members use these members to test the buffer's contents.

isDBCSLead If the input character is the first byte of a DBCS character, true is returned.

| <pre>static Boolean isDBCSLead(char inByte);</pre> | <table><thead><tr><th><u>Win</u></th><th><u>PM</u></th><th><u>Motif</u></th></tr></thead><tbody><tr><td><i>Y</i></td><td><i>Y</i></td><td><i>N</i></td></tr></tbody></table> | <u>Win</u> | <u>PM</u> | <u>Motif</u> | <i>Y</i> | <i>Y</i> | <i>N</i> |
|--|--|--------------|-----------|--------------|----------|----------|----------|
| <u>Win</u> | <u>PM</u> | <u>Motif</u> | | | | | |
| <i>Y</i> | <i>Y</i> | <i>N</i> | | | | | |

Public Data

DBCS Table

Use this character array member to test characters for DBCS validity.

dbcsTable Provides a table of DBCS first-byte flags ('dbcsTable[n] == 1' if n is a valid DBCS first byte).

| <pre>static char dbcsTable [256];</pre> | <table><thead><tr><th><u>Win</u></th><th><u>PM</u></th><th><u>Motif</u></th></tr></thead><tbody><tr><td><i>Y</i></td><td><i>Y</i></td><td><i>N</i></td></tr></tbody></table> | <u>Win</u> | <u>PM</u> | <u>Motif</u> | <i>Y</i> | <i>Y</i> | <i>N</i> |
|---|--|--------------|-----------|--------------|----------|----------|----------|
| <u>Win</u> | <u>PM</u> | <u>Motif</u> | | | | | |
| <i>Y</i> | <i>Y</i> | <i>N</i> | | | | | |

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. 341).

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
IBaseErrorInfo
ICLibErrorInfo

Inherited By None.

Header File

iexcept.hpp

Members

| Member | Page | Member | Page |
|-----------------------|------|-----------------|------|
| Constructor | 359 | text | 360 |
| errorId | 359 | throwCLibError | 360 |
| isAvailable | 359 | ~ICLibErrorInfo | 359 |
| operator const char * | 359 | | |

The ICLibErrorInfo class represents 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 User Interface Class 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 IBaseErrorInfo::ExceptionType (p. 319) to specify the type of the exception. The default is accessError.
- severity* Use the enumeration IException::Severity (p. 403) to specify the severity of the error. The default is recoverable.

This macro generates code that calls throwCLibError (p. 360), 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

Constructors

You can construct and destruct objects of this class. You cannot copy or assign objects of this class.

ICLibErrorInfo

```
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 User Interface Class Library provides the following default text: "No error text is available."

CLibFunctionName

The name of the failing C library function. If you specify *CLibFunctionName*, the constructor prefixes it to the error text. Optional.

~ICLibErrorInfo

```
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;
```

ICLibErrorInfo

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 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. 406) object containing the following:

- Function name
- File name
- Line number where the function is called

name Use the enumeration IBaseErrorInfo::ExceptionType (p. 319) to specify the type of the exception. The default is accessError.

severity Use the enumeration IException::Severity (p. 403) 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

| IBaseErrorInfo | | |
|----------------|-------------|-----------------------|
| errorId | isAvailable | operator const char * |

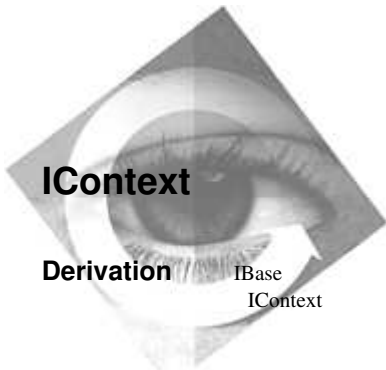
ICLibErrorInfo

| IVBase | | |
|-------------|----------|--|
| asDebugInfo | asString | |

| IBase | | |
|-------------|-------------|----------------|
| asDebugInfo | messageFile | setMessageFile |
| asString | messageText | version |

Inherited Protected Data

| IBase | | |
|-------------|---------------|--|
| recoverable | unrecoverable | |



IContext

Derivation IBase
IContext

Inherited By None.

Header File
icontext.hpp

| Members | Member | Page | Member | Page |
|---------|-------------|------|-----------|------|
| | Constructor | 362 | find | 363 |
| | add | 363 | reset | 363 |
| | count | 363 | ~IContext | 363 |

The IContext class provides a method of determining the object when multiple references to the same object are flattened and packed together on the same stream. This class is basically a dynamic dictionary.

Objects of this class can be built during the flattening process to assign references to repeated object instances from a set of objects to be saved to a stream. If you only want to save a single instance then a default context of the stream is used automatically throughout the flattening process, which guarantees that only one instance of an object is saved, even though the flatten function is called multiple times.

If you want to save multiple instances (which might point to overlapping instances), you need to set the stream with different IContext objects during the flattening process.

Public Functions

Constructors

Use these members to construct and delete IContext objects.

IContext Use these member functions to construct IContext objects.

| | | | |
|-------------|------------|-----------|--------------|
| IContext(); | <u>Win</u> | <u>PM</u> | <u>Motif</u> |
| | Y | N | N |

IContext

Use this function to construct a default IContext object.

~IContext

| | | | |
|-------------------------|-----------------|----------------|-------------------|
| virtual ~IContext(); | <u>Win</u> Y | <u>PM</u> N | <u>Motif</u> N |
|-------------------------|-----------------|----------------|-------------------|

Destroys an IContext object.

Dictionary Access

Use these members to access objects in the IContext dictionary.

add

Adds an object to the dictionary of this IContext object. The return value represents the key to the object in the dictionary. It can be used to retrieve the object using the find function.

The second parameter is for both in and out. On the way in, if it is true, a new entry in the dictionary is created for the object even if it already exists in the dictionary. (In this case, the old entry is deleted first.) On the way out, this parameter returns if a new entry has been created for the object. If the input value of the second parameter is false and an entry of the object already exists, this function returns the number that is the key to the existing entry of the object.

| | | | |
|--|-----------------|----------------|-------------------|
| ObjectNumber add(const void* object, Boolean& objectExistsFlag); | <u>Win</u> Y | <u>PM</u> N | <u>Motif</u> N |
|--|-----------------|----------------|-------------------|

count

Obtains the context count.

| | | | |
|---------------------------|-----------------|----------------|-------------------|
| unsigned long count(); | <u>Win</u> Y | <u>PM</u> N | <u>Motif</u> N |
|---------------------------|-----------------|----------------|-------------------|

find

Returns the object in the dictionary of this IContext object that has the key equal to the input parameter.

| | | | |
|---|-----------------|----------------|-------------------|
| const void* find(ObjectNumber objectNumber); | <u>Win</u> Y | <u>PM</u> N | <u>Motif</u> N |
|---|-----------------|----------------|-------------------|

reset

Resets the dictionary in the IContext object to empty.

| | | | |
|-----------------------|-----------------|----------------|-------------------|
| IContext& reset(); | <u>Win</u> Y | <u>PM</u> N | <u>Motif</u> N |
|-----------------------|-----------------|----------------|-------------------|

IContext

Inherited Public Functions

| IBase | | |
|-------------|-------------|----------------|
| asDebugInfo | messageFile | setMessageFile |
| asString | messageText | version |

Inherited Protected Data

| IBase | | |
|-------------|---------------|--|
| recoverable | unrecoverable | |

Nested Type Definitions

ObjectNumber

```
typedef unsigned long ObjectNumber;
```

Object number is the data type of the keys of the objects in the dictionary for an IContext object. It is defined as an unsigned long data type.



IDate

Derivation IBase
IDate

Inherited By None.

Header File
idate.hpp

| Members | Member | Page | Member | Page |
|---------|-------------|------|-------------|------|
| | Constructor | 367 | monthOfYear | 372 |
| | asICnrDate | 368 | operator != | 366 |
| | asString | 369 | operator + | 371 |
| | dayName | 368 | operator += | 371 |
| | dayOfMonth | 368 | operator - | 372 |
| | dayOfWeek | 368 | operator -= | 372 |
| | dayOfYear | 369 | operator < | 366 |
| | daysInMonth | 371 | operator <= | 366 |
| | daysInYear | 371 | operator == | 366 |
| | initialize | 374 | operator > | 366 |
| | isLeapYear | 373 | operator >= | 366 |
| | isValid | 373 | today | 368 |
| | julianDate | 368 | year | 373 |
| | monthName | 371 | | |

The IDate class represents 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 User Interface Class 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 OS/2 documentation for information about setting the locale.

Public Functions

IDate

Comparisons

Use these members to compare two IDate's. Use any of the full complement of comparison operators and apply 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.

IDate

- Give the year and day of the year for the desired day.
- Use `IDate::today` (p. 368) to return the current date.
- Copy another `IDate` object.
- Give the Julian day number, as a long.
- Give a container details `CDATE` structure.

IDate

1 `IDate();`

Use this constructor to return the current day; it's the default.

2 `IDate(Month aMonth,
 int aDay,
 int aYear);`

Use this constructor to pass parameters in month/day/year order.

3 `IDate(int aDay,
 Month aMonth,
 int aYear);`

Use this constructor when passing parameters in day/month/year order.

4 `IDate(int aYear,
 int aDay);`

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);`

Constructs an `IDate` by copying another `IDate` object.

6 `IDate(unsigned long julianDayNumber);`

Constructs an `IDate` from a Julian day number, as a long.

7 `IDate(const ICnrDate& cnrDate);`

Constructs an `IDate` from a container details `ICnrDate` structure.

IDate

Conversions

Use these members to retrieve other representations of the date.

asICnrDate Returns a container ICnrDate structure for the date.

```
ICnrDate  
asICnrDate() const;
```

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 following these criteria:

- 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.

```
IStrng  
dayName() const;
```

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.

IDate

```
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 have 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. 586) provides the conversion specifiers that apply to ITime. For more information about the strftime function, refer to the *VisualAge for C++: C Library Reference*.

IDate

| 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")
```

- 1 IString
asString(const char* fmt) const;
- 2 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 following these criteria:

- 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*.

IDate

- 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 );
```

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 following these criteria:

- 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 );
```

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 );
```

IDate

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.

1 IDate
operator -(int numDays) const;

2 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 following these criteria:

- 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;

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 follows:

- 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.

```
1 static Boolean
    isValid( int aDay,
            Month aMonth,
            int aYear );
```

```
2 static Boolean
    isValid( Month aMonth,
            int aDay,
            int aYear );
```

```
3 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;
```

IDate

Inherited Public Functions

| IBase | | |
|-------------|-------------|----------------|
| asDebugInfo | messageFile | setMessageFile |
| asString | messageText | version |

Protected Functions

Implementation

These members initialize 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

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.

IDate

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.

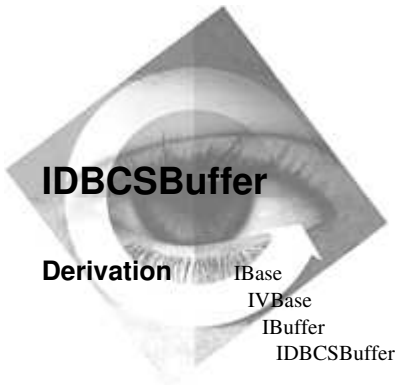
YearFormat `typedef enum { yy , yyyy } YearFormat;`

A typedef that specifies the number of digits in the year for the default asString format (yy or yyyy).

Associated Globals

operator << `ostream& operator <<(ostream& aStream, const IDate& aDate);`

Outputs the asString representation of *aDate*.



IDBCSBuffer

Derivation

IBase
IVBase
IBuffer
IDBCSBuffer

Inherited By None.

Header File

idbcsbuf.hpp

Members

| Member | Page | Member | Page |
|----------------|------|----------------------|------|
| Constructor | 388 | isPrintable | 386 |
| allocate | 377 | isPunctuation | 386 |
| center | 377 | isSBC | 390 |
| change | 377 | isSBCS | 381 |
| charLength | 388 | isUpperCase | 386 |
| charType | 381 | isValidDBCS | 381 |
| className | 388 | isValidMBCS | 381 |
| compare | 377 | isWhiteSpace | 386 |
| includesDBCS | 380 | lastIndexOf | 382 |
| includesMBCS | 380 | lastIndexOfAnyBut | 383 |
| includesSBCS | 380 | lastIndexOfAnyOf | 383 |
| indexOf | 381 | leftJustify | 378 |
| indexOfAnyBut | 382 | lowerCase | 378 |
| indexOfAnyOf | 382 | maxCharLength | 389 |
| insert | 378 | next | 381 |
| isAlphabetic | 385 | overlayWith | 378 |
| isAlphanumeric | 385 | prevCharLength | 389 |
| isASCII | 385 | remove | 379 |
| isCharValid | 389 | reverse | 379 |
| isControl | 385 | rightJustify | 379 |
| isDBCS | 380 | startBackwardsSearch | 390 |
| isDBCS1 | 390 | startSearch | 391 |
| isDigits | 385 | strip | 379 |
| isGraphics | 385 | subString | 384 |
| isHexDigits | 385 | translate | 380 |
| isLowerCase | 386 | upperCase | 380 |
| isMBCS | 381 | ~IDBCSBuffer | 388 |
| isPrevDBCS | 390 | | |

The IDBCSBuffer class implements the version of IString (p. 508) 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.

```
virtual IBuffer*
    allocate( unsigned newLen ) const;
```

Comparisons

Use these members to compare the IDBCSBuffer'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;
```

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.

```
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 a NULL-terminated string. The library searches for the pattern string within the receiver's data.

sourceLen The length of the source string.

IDBCSBuffer

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 );
```

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 (*length*) 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 uppercase letters in the receiver to lowercase.

```
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 (*length*), 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 the leading and trailing character or characters. You can specify the character or characters as the following:

- A char* array
- An IStringTest (p. 573) object

The default is white space.

```
1 virtual IBuffer*
    strip( const char* pChars,
           unsigned len,
           IStringEnum::StripMode mode );
```

```
2 virtual IBuffer*
    strip( const IStringTest& aTest,
           IStringEnum::StripMode mode );
```

IDBCSBuffer

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 lowercase letters in the receiver to uppercase.

```
virtual 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.

```
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;
```

IDBCSBuffer

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 reimplement the following IString versions of IBuffer members.

charType Returns the type of a character at the specified index.

```
virtual IStringEnum::CharType  
    charType( unsigned index ) const;
```

next Returns a pointer to the next character, not the next byte, in the buffer.

```
1 virtual const char*  
    next( const char* prev ) const;
```

```
2 virtual 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.

IDBCSBuffer

- 1** virtual unsigned
 indexOf(const IStringTest& aTest,
 unsigned startPos = 1) const;
- 2** virtual 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. 573) object.

- 1** virtual unsigned
 indexOfAnyBut(const IStringTest& aTest,
 unsigned startPos = 1) const;
- 2** 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. 573) object.

- 1** virtual unsigned
 indexOfAnyOf(const char* pString,
 unsigned len,
 unsigned startPos = 1) const;
- 2** 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 $0 \leq x \leq startPos$. 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 1 for *startPos*, this function returns 0 indicating the search target was not found.

```
1 virtual unsigned
  lastIndexOf( const char* pString,
               unsigned len,
               unsigned startPos = 0 ) const;
```

```
2 virtual 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.

```
1 virtual unsigned
  lastIndexOfAnyBut( const char* pString,
                    unsigned len,
                    unsigned startPos = 0 ) const;
```

```
2 virtual 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.

IDBCSBuffer

```
1 virtual unsigned
  lastIndexOfAnyOf( const char* pString,
                    unsigned len,
                    unsigned startPos = 0 ) const;
```

```
2 virtual 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.

```
1 virtual IBuffer*
  subString( unsigned startPos ) const;
```

```
2 virtual IBuffer*
  subString( unsigned startPos,
            unsigned len,
            char padCharacter ) const;
```

Testing

Corresponding IString members use these members to test the buffer's contents.

Note: In the Windows environment, the Testing functions are implemented using the wide char functions of the C runtime library.

IDBCSBuffer

On other platforms, IDBCSBuffer inherits the Testing functions from the IBuffer (p. 340) class.

isAlphabetic If all the characters are in {'A'-'Z','a'-'z'}, true is returned.

```
virtual Boolean  
    isAlphabetic() const;
```

isAlphanumeric

If all 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 `iswcntrl()` C Library function as defined in the `cntrl` locale source file 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 `iswgraph()` 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.

IDBCSBuffer

```
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 `iswprint()` 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 white-space characters.

White-space characters are defined by the `iswspace()` 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 white-space characters are those in the range {0x09-0x0D,0x20}.

```
virtual Boolean  
    isWhiteSpace() const;
```

Inherited Public Functions

| IBuffer | | |
|----------------------------|-------------------|-------------------------|
| addRef | isAlphabetic | lastIndexOfAnyOf |
| asDebugInfo | isAlphanumeric | leftJustify |
| center | isASCII | length |
| change | isControl | lowerCase |
| 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

Constructors

The constructor for this class is protected. Only IDBCSBuffer::allocate (p. 377) and IBuffer::initialize (p. 355) can call the constructor.

IDBCSBuffer

IDBCSBuffer

```
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. IDBCSBuffer objects must be obtained by using IDBCSBuffer::nullBuffer and subsequent newBuffer calls to existing IDBCSBuffer objects.

~IDBCSBuffer

```
~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.

```
1 size_t  
  charLength( unsigned pos,  
              mbstate_t* pMBState ) const;
```

```
2 static size_t  
  charLength( char const* pChar );
```

```
3 size_t  
  charLength( unsigned pos ) const;
```

```
4 static size_t  
  charLength( char const* pChar,  
              mbstate_t* pMBState );
```

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.

```
1 size_t
  prevCharLength( unsigned pos,
                  mbstate_t* pMBState ) const;
```

```
2 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;
```

IDBCSBuffer

isDBCS1 If the byte at the specified offset is the first byte of DBCS, true is returned.

Note: The User Interface Class Library provides this function only for compatibility with prior library versions. We recommend using IDBCSBuffer::charLength (p. 388) to determine if the byte is part of a multiple-byte character.

| | | | |
|--------------------------------|------------|-----------|--------------|
| Boolean | Win | PM | Motif |
| isDBCS1(unsigned pos) const; | <i>N</i> | <i>Y</i> | <i>N</i> |

isPrevDBCS If the preceding character to the one at the specified offset is a DBCS character, true is returned.

Note: The User Interface Class Library provides this function only for compatibility with prior library versions. We recommend using IDBCSBuffer::prevCharLength (p. 389) to determine if the preceding byte is part of a multiple-byte character.

| | | | |
|-----------------------------------|------------|-----------|--------------|
| Boolean | Win | PM | Motif |
| isPrevDBCS(unsigned pos) const; | <i>N</i> | <i>Y</i> | <i>N</i> |

isSBC If the byte pointed to by the specified character is a single-byte character, true is returned. This is a static function.

1 static Boolean
isSBC(char const* pChar,
mbstate_t* pMBState);

2 static Boolean
isSBC(char const* pChar);

Search Initialization

These members initialize search data.

startBackwardsSearch

Initializes a search of type IString::lastIndexOf (p. 539) following these criteria:

- 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.

IDBCSBuffer

```
virtual unsigned  
    startBackwardsSearch( unsigned startPos,  
                          unsigned searchLen ) const;
```

startSearch Initializes a search of type `IString::indexOf` (p. 525) following these criteria:

- 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;
```

Inherited Protected Functions

| IBuffer | | |
|-----------|------------|-----------------|
| allocate | initialize | operator delete |
| className | isDBCSLead | operator new |

Inherited Public Data

| IBuffer | | |
|-----------|--|--|
| dbcsTable | | |

Inherited Protected Data

| IBase | | |
|-------------|---------------|--|
| recoverable | unrecoverable | |



IODeviceError

Derivation IException
IODeviceError

Inherited By None.

Header File
iexcbase.hpp

| Members | Member | Page |
|---------|-------------|------|
| | Constructor | 392 |
| | name | 393 |

The IDeviceError class represents 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

Constructors

You can construct objects of this class.

IODeviceError 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. 403) to specify the severity of the error. The default is unrecoverable.
- Using the macros discussed in IException (p. 394). The User Interface Class Library provides these macros to make creating exceptions easier for you.

```
IODeviceError( const char* errorText,  
               unsigned long errorId,  
               Severity severity = IException::unrecoverable );
```

Exception Type

These members provide support for determining the name (type) of the exception. They are used for logging out an exception object's error information.

name Returns the name of the object's class.

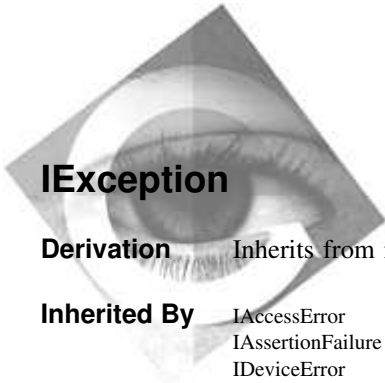
```
virtual const char*
name() const;
```

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 |



IException

Derivation Inherits from none.

Inherited By IAccessError IAssertionFailure IDeviceError IInvalidParameter IInvalidRequest IResourceExhausted

Header File
iexcbase.hpp

| Members | | Member | Page | Member | Page |
|---------|--|------------------|------|--------------------|------|
| | | Constructor | 397 | operatingSystem | 402 |
| | | addLocation | 398 | other | 402 |
| | | appendText | 400 | presentationSystem | 402 |
| | | assertParameter | 401 | setErrorCodeGroup | 398 |
| | | baseLibrary | 401 | setErrorId | 398 |
| | | CLibrary | 402 | setSeverity | 400 |
| | | errorCodeGroup | 397 | setText | 400 |
| | | errorId | 398 | setTraceFunction | 399 |
| | | isRecoverable | 400 | terminate | 396 |
| | | locationAtIndex | 398 | text | 400 |
| | | locationCount | 399 | textCount | 400 |
| | | logExceptionData | 399 | ~IException | 397 |
| | | name | 401 | | |

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 User Interface Class 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.

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:

IException

- IAccessError (p. 304)
- IAssertionFailure (p. 306)
- IDeviceError (p. 392)
- IInvalidParameter (p. 418)
- IInvalidRequest (p. 420)
- IResourceExhausted (p. 498)

In addition, IResourceExhausted has the following derived classes:

- IOutOfMemory (p. 453)
- IOutOfSystemResource (p. 455)
- IOutOfWindowResource (p. 457)

You can also derive your own exception type from IException.

The User Interface Class 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 rethrowing 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. 401).

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. 406).

INO_EXCEPTIONS_SUPPORT

Supports compilers lacking an exception-handling implementation. If you use the INO_EXCEPTIONS_SUPPORT macro, the following macros end the

IException

program after capturing the location information and logging it. These macros normally throw an exception:

| | |
|----------------------------|-------------------------------------|
| ITHROW | Found in IException. |
| IASSERTPARG | Found in IBaseErrorInfo (p. 314). |
| IASSERTSTATE | Found in IBaseErrorInfo. |
| ITHROWERROR | Found in IBaseErrorInfo. |
| ITHROWERROR1 | Found in IBaseErrorInfo. |
| ITHROWLIBRARYERROR | Found in IBaseErrorInfo. |
| ITHROWLIBRARYERROR1 | Found in IBaseErrorInfo. |
| ITHROWGUIERROR | Found in IGUIErrorInfo (p. 413). |
| ITHROWGUIERROR2 | Found in IGUIErrorInfo. |
| ITHROWSYSTEMERROR | Found in ISystemErrorInfo (p. 580). |

Warning: The INO_EXCEPTIONS_SUPPORT macro might not work correctly on all compilers.

Whenever the User Interface Class Library throws one of these exceptions, trace records are output with information about the exception. The class ITrace (p. 590) 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 User Interface Class 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 as follows:

| | |
|----------------------------|-------------------------------------|
| ITHROW | Found in IException. |
| IASSERTPARG | Found in IBaseErrorInfo (p. 314). |
| IASSERTSTATE | Found in IBaseErrorInfo. |
| ITHROWLIBRARYERROR | Found in IBaseErrorInfo. |
| ITHROWLIBRARYERROR1 | Found in IBaseErrorInfo. |
| ITHROWGUIERROR | Found in IGUIErrorInfo (p. 413). |
| ITHROWGUIERROR2 | Found in IGUIErrorInfo. |
| ITHROWSYSTEMERROR | Found in ISystemErrorInfo (p. 580). |

```
virtual void  
    terminate();
```

Constructors

You can construct and destruct objects of this class. You cannot assign one IException object from another.

IException

```
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 The identifier you want to associate with this particular error. Optional.

severity Use the enumeration IException::Severity (p. 403) to specify the severity of the error. The default is unrecoverable. Optional.

- Using the copy constructor. The User Interface Class 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.

~IException

```
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.

IException

```
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 User Interface Class Library captures this information when an exception is thrown or rethrown. An array of IExceptionLocation objects is stored in the exception object.

location An IExceptionLocation (p. 406) 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. 406) object at the specified index.

IException

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. 399). 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. 404) to be used to log exception data. The ITrace (p. 590) member functions and macros write the trace messages. IException::logExceptionData (p. 399) calls IException::TraceFn::write (p. 405) 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.

IException

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. 403) 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.

IException

```
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. 306) 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. 406) 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 Specifies the error group for IBM Open Class Library errors.

IException

```
static ErrorCodeGroup const  
    baseLibrary;
```

CLibrary Specifies the error group for the C library errors.

```
static ErrorCodeGroup const  
    CLibrary;
```

operatingSystem

Specifies the error group for operating system errors.

```
static ErrorCodeGroup const  
    operatingSystem;
```

other Specifies the error group for errors which do not fall in any of the other groups.

```
static ErrorCodeGroup const  
    other;
```

presentationSystem

Specifies the error group for presentation system errors.

```
static ErrorCodeGroup const  
    presentationSystem;
```

Nested Classes

IException contains the following nested classes:

IException::TraceFn (see page 404)

Nested Type Definitions

Severity Severity {
 unrecoverable,
 recoverable
 };

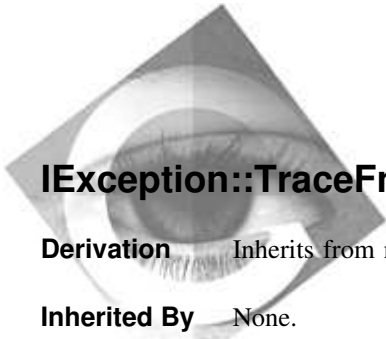
Use these enumerators to specify the severity of the exception:

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 | 405 | TraceFn | 405 |
| exceptionLogged | 405 | write | 405 |
| logData | 405 | | |

Objects of the class IException (p. 394) 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. 590).

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. 399). 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 want 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 out all of the exception data 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 IException 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.

TraceFn This default constructor can be used by derived classes to create objects of this class.

```
TraceFn();
```

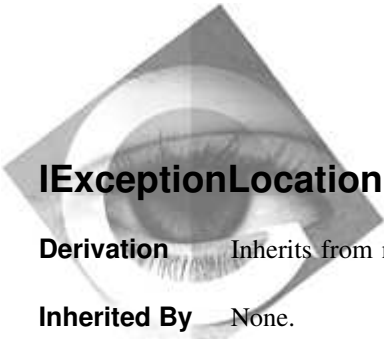
Tracing

The IException's logExceptionData member 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 | 407 | functionName | 406 |
| | fileName | 406 | lineNumber | 406 |

The IExceptionLocation class saves 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 rethrown.

```
const char*
  fileName() const;
```

functionName Returns the name of the function that has thrown or rethrown 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 rethrown.

IExceptionLocation

```
unsigned long  
lineNumber() const;
```

Constructors

You can construct objects of this class.

IExceptionLocation

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. 394). 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 );
```



Inherited By None.

Header File
ifilstrm.hpp

| Members | Member | Page | Member | Page |
|---------|-----------------------|------|------------------------|------|
| | Constructor | 408 | physicalEndOfStream | 409 |
| | flush | 409 | position | 409 |
| | handleReadBufferEmpty | 411 | seek | 409 |
| | handleWriteBufferFull | 411 | setLogicalEndOfStream | 410 |
| | isWriteable | 409 | setPhysicalEndOfStream | 411 |
| | logicalEndOfStream | 409 | ~IFileStream | 409 |

The IFileStream class is a concrete class derived from IBaseStream that lets you read or write a file via a stream.

Public Functions

Constructors

You can construct and destruct objects of this class. You cannot copy or assign objects of this class.

IFileStream Constructs objects of the IFileStream class.

```
IFileStream( const char* fileName,
             Boolean writeable = true );
```

| | | |
|------------|-----------|--------------|
| <u>Win</u> | <u>PM</u> | <u>Motif</u> |
| Y | N | N |

Use this function to construct an IFileStream for a specified file. If the second parameter is set to false, the file is opened in read-only mode.

The parameters are the following:

fileName The file to be opened.
writeable Indicates if the file can be written to.

IFileStream

| Exceptions | |
|--------------|----------------------------|
| IAccessError | The file cannot be opened. |

~IFileStream

| | | | | |
|-----------------|--|------------|-----------|--------------|
| virtual | | <u>Win</u> | <u>PM</u> | <u>Motif</u> |
| ~IFileStream(); | | <i>Y</i> | <i>N</i> | <i>N</i> |

Stream Positioning, Flushing, and Write Access

These members control the buffering, write access, and positioning of a file stream.

flush Flushes buffered changes to the disk.

| | | | |
|----------------------|------------|-----------|--------------|
| virtual IFileStream& | <u>Win</u> | <u>PM</u> | <u>Motif</u> |
| flush(); | <i>Y</i> | <i>N</i> | <i>N</i> |

isWriteable Returns true if the file stream is writeable.

| | | | |
|----------------------|------------|-----------|--------------|
| virtual Boolean | <u>Win</u> | <u>PM</u> | <u>Motif</u> |
| isWriteable() const; | <i>Y</i> | <i>N</i> | <i>N</i> |

logicalEndOfStream

Obtains the logical size of the file stream.

| | | | |
|-------------------------------|------------|-----------|--------------|
| virtual IBaseStream::Position | <u>Win</u> | <u>PM</u> | <u>Motif</u> |
| logicalEndOfStream() const; | <i>Y</i> | <i>N</i> | <i>N</i> |

physicalEndOfStream

Obtains the physical size of the file stream.

| | | | |
|-------------------------------|------------|-----------|--------------|
| virtual IBaseStream::Position | <u>Win</u> | <u>PM</u> | <u>Motif</u> |
| physicalEndOfStream() const; | <i>Y</i> | <i>N</i> | <i>N</i> |

position Obtains the current read/write pointer for the stream.

| | | | |
|-------------------------------|------------|-----------|--------------|
| virtual IBaseStream::Position | <u>Win</u> | <u>PM</u> | <u>Motif</u> |
| position() const; | <i>Y</i> | <i>N</i> | <i>N</i> |

seek Sets the read/write pointer for the stream to *position*.

IFileStream

```
virtual IFileStream&  
    seek( IBaseStream::Position position );
```

| <u>Win</u> | <u>PM</u> | <u>Motif</u> |
|------------|-----------|--------------|
| <i>Y</i> | <i>N</i> | <i>N</i> |

| Exceptions | |
|-------------------|---|
| IInvalidParameter | The seek position in the stream is not valid. |

setLogicalEndOfStream

Sets the logical size of the file stream to *position*.

```
virtual IFileStream&  
    setLogicalEndOfStream( IBaseStream::Position position );
```

| <u>Win</u> | <u>PM</u> | <u>Motif</u> |
|------------|-----------|--------------|
| <i>Y</i> | <i>N</i> | <i>N</i> |

| Exceptions | |
|--------------|---|
| IAccessError | The file cannot be opened. |
| IAccessError | There is insufficient space in the file system. |

Inherited Public Functions

| IBaseStream | | |
|----------------|---------------------|-----------------------|
| context | logicalEndOfStream | seek |
| defaultContext | physicalEndOfStream | seekRelative |
| flush | position | setContext |
| isWriteable | read | setLogicalEndOfStream |

| IVBase | | |
|-------------|----------|--|
| asDebugInfo | asString | |

| IBase | | |
|-------------|-------------|----------------|
| asDebugInfo | messageFile | setMessageFile |
| asString | messageText | version |

Protected Functions

Buffer Management

These members implement the reading and writing of data to a file stream.

IFileStream

handleReadBufferEmpty

Handles reading data from the file stream.

| | | | |
|---|-----------------|----------------|-------------------|
| virtual void handleReadBufferEmpty(void* target, size_t byteCount); | <u>Win</u> Y | <u>PM</u> N | <u>Motif</u> N |
|---|-----------------|----------------|-------------------|

The parameters are the following:

target The address to store the data that is read.
byteCount The number of bytes to be read.

| Exceptions | |
|------------------|---|
| InvalidParameter | There is insufficient data in the file. |

handleWriteBufferFull

Handles writing of data to the file stream.

| | | | |
|---|-----------------|----------------|-------------------|
| virtual void handleWriteBufferFull(const void* source, size_t byteCount); | <u>Win</u> Y | <u>PM</u> N | <u>Motif</u> N |
|---|-----------------|----------------|-------------------|

The parameters are the following:

source The address of the data to write.
byteCount The number of bytes to be written.

| Exceptions | |
|------------------|---|
| InvalidParameter | An error occurred writing the stream to the file. |

Stream Positioning, Flushing, and Write Access

These members control the buffering, write access, and positioning of a file stream.

setPhysicalEndOfStream

Sets the physical size of the file stream to *position*.

| | | | |
|---|-----------------|----------------|-------------------|
| virtual IFileStream& setPhysicalEndOfStream(IBaseStream::Position position); | <u>Win</u> Y | <u>PM</u> N | <u>Motif</u> N |
|---|-----------------|----------------|-------------------|

| Exceptions | |
|--------------|---|
| IAccessError | The file cannot be opened. |
| IAccessError | There is insufficient space in the file system. |

IFileStream

Inherited Protected Functions

| IBaseStream | | |
|-----------------------|-----------------------|--|
| handleReadBufferEmpty | handleWriteBufferFull | |

Inherited Protected Data

| IBase | | |
|-------------|---------------|--|
| recoverable | unrecoverable | |



IGUIErrorInfo

Derivation

- IBase
- IVBase
- IBaseErrorInfo
- IGUIErrorInfo

Inherited By None.

Header File

iexcept.hpp

| Members | Member | Page | Member | Page |
|---------|-----------------------|------|----------------|------|
| | Constructor | 414 | text | 415 |
| | errorId | 415 | throwGUIError | 416 |
| | isAvailable | 415 | ~IGUIErrorInfo | 415 |
| | operator const char * | 415 | | |

The IGUIErrorInfo class represents 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. 394).

The User Interface Class 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. 416), which does the following:

1. Creates an IGUIErrorInfo object
2. Uses the object to create an object of IAccessError (p. 304)
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. 403).

ITHROWGUIERROR2

This macro can throw any of the User Interface Class Library-defined exceptions. This macro accepts the following parameters:

IGUIErrorInfo

| | |
|-----------------|--|
| <i>location</i> | 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. |
| <i>name</i> | Use the enumeration IBaseErrorInfo::ExceptionType (p. 319) to specify the type of the exception. The default is accessError. |
| <i>severity</i> | Use the enumeration IException::Severity (p. 403) to specify the severity of the error. The default is recoverable. |

This macro generates code that calls throwGUIError (p. 416), 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



You can use objects of the IGUIErrorInfo class to obtain information about the last error that occurred on a call to Presentation Manager.



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

Constructors

You can construct and destruct objects of this class. You cannot copy or assign objects of this class.

IGUIErrorInfo

```
IGUIErrorInfo( const char* GUIFunctionName = 0 );
```

You can only construct objects of this class using the default constructor.

IGUIErrorInfo

Note: If the constructor cannot load the error text, the User Interface Class 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.

~IGUIErrorInfo

```
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;
```



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.

IGUIErrorInfo

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. 406) object containing the following:

- Function name
- File name
- Line number where the function is called

name Use the enumeration IBaseErrorInfo::ExceptionType (p. 319) to specify the type of the exception. The default is accessError.

severity Use the enumeration IException::Severity (p. 403) 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

| IBaseErrorInfo | | |
|----------------|-------------|-----------------------|
| errorId | isAvailable | operator const char * |

| IVBase | | |
|-------------|----------|--|
| asDebugInfo | asString | |

| IBase | | |
|-------------|-------------|----------------|
| asDebugInfo | messageFile | setMessageFile |
| asString | messageText | version |

Inherited Protected Data

| IBase | | |
|-------------|---------------|--|
| recoverable | unrecoverable | |



Derivation IException
InvalidParameter

Inherited By None.

Header File
iexcbase.hpp

| Members | Member | Page |
|---------|-------------|------|
| | Constructor | 418 |
| | name | 419 |

The InvalidParameter class represents an exception. When a member function detects an invalid input parameter, the member function creates and throws an object of the InvalidParameter class. This exception is identical to the exception IAssertionFailure (p. 306), with one difference: InvalidParameter is thrown whether or not you define IC_DEVELOP for the compile.

Public Functions

Constructors

You can construct objects of this class.

InvalidParameter

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. 403) to specify the severity of the error. The default is unrecoverable.
- Using the macros discussed in IException (p. 394). The User Interface Class Library provides these macros to make creating exceptions easier for you.

IInvalidParameter

```
IInvalidParameter(  
    const char* errorText,  
    unsigned long errorId,  
    Severity severity = IException::unrecoverable );
```

Exception Type

Use these members to determine the name (type) of the exception. They are 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 |



InvalidRequest

Derivation IException
InvalidRequest

Inherited By None.

Header File
iexcbase.hpp

| Members | Member | Page |
|---------|-------------|------|
| | Constructor | 420 |
| | name | 421 |

The InvalidRequest class represents an exception. Whenever an object cannot satisfy a request, the member function creates and throws an object of the InvalidRequest 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

Constructors

You can construct objects of this class.

InvalidRequest

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. 403) to specify the severity of the error. The default is unrecoverable.
- Using the macros discussed in IException (p. 394). The User Interface Class Library provides these macros to make creating exceptions easier for you.

IInvalidRequest

```
IInvalidRequest(  
    const char* errorText,  
    unsigned long errorId,  
    Severity severity = IException::unrecoverable );
```

Exception Type

Use these members to determine the name (type) of the exception. They are used for logging out an exception object's error information.

name Returns the name of the object's class.

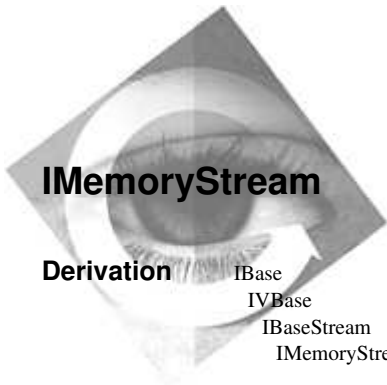
```
virtual const char*  
    name() const;
```

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 |



IMemoryStream

Derivation

IBase
IVBase
IBaseStream
IMemoryStream

Inherited By None.

Header File

imemstrm.hpp

| Members | Member | Page | Member | Page |
|---------|-----------------------|------|------------------------|------|
| | Constructor | 423 | position | 423 |
| | buffer | 422 | seek | 423 |
| | handleReadBufferEmpty | 424 | setPhysicalEndOfStream | 425 |
| | handleWriteBufferFull | 425 | ~IMemoryStream | 423 |
| | physicalEndOfStream | 423 | | |

The IMemoryStream class is a concrete class derived from IBaseStream that supports reading from and writing to contiguous memory on the heap. The size of the memory can automatically grow when needed.

Public Functions

Buffer Management

Use these members to implement the reading and writing of data to a memory stream.

buffer Returns a pointer to the internal buffer of this memory stream object.

| | | | |
|-------------|------------|-----------|--------------|
| const char* | <u>Win</u> | <u>PM</u> | <u>Motif</u> |
| buffer(); | Y | N | N |

Constructors

You can construct and destruct objects of this class. You cannot copy or assign objects of this class.

IMemoryStream

IMemoryStream

Constructs IMemoryStream objects.

| | | | | |
|----------|-------------------------------|-------------------|------------------|---------------------|
| 1 | <code>IMemoryStream();</code> | <u>Win</u> | <u>PM</u> | <u>Motif</u> |
| | | <i>Y</i> | <i>N</i> | <i>N</i> |

Use this function to construct a default memory stream object.

| | | | | |
|----------|---|-------------------|------------------|---------------------|
| 2 | <code>IMemoryStream(const void* memory, size_t byteCount);</code> | <u>Win</u> | <u>PM</u> | <u>Motif</u> |
| | | <i>Y</i> | <i>N</i> | <i>N</i> |

Use this function to construct an IMemoryStream object to copy the specified number of bytes from the input address to the buffer of the IMemoryStream object. The position of the stream is set to the first byte after the written data.

The parameters are the following:

| | |
|---------------|--|
| <i>memory</i> | The address of the data to be written to the stream. |
| <i>size</i> | The number of bytes to be written. |

~IMemoryStream

| | | | |
|--|-------------------|------------------|---------------------|
| <code>virtual ~IMemoryStream();</code> | <u>Win</u> | <u>PM</u> | <u>Motif</u> |
| | <i>Y</i> | <i>N</i> | <i>N</i> |

Stream Positioning

These members control the positioning of a file stream.

physicalEndOfStream

Obtains the physical size of the memory stream.

| | | | |
|---|-------------------|------------------|---------------------|
| <code>virtual IBaseStream::Position physicalEndOfStream() const;</code> | <u>Win</u> | <u>PM</u> | <u>Motif</u> |
| | <i>Y</i> | <i>N</i> | <i>N</i> |

position

Obtains the current read/write pointer for the stream.

| | | | |
|--|-------------------|------------------|---------------------|
| <code>virtual IBaseStream::Position position() const;</code> | <u>Win</u> | <u>PM</u> | <u>Motif</u> |
| | <i>Y</i> | <i>N</i> | <i>N</i> |

seek

Sets the read/write pointer for the stream to *position*.

IMemoryStream

```
virtual IMemoryStream&
    seek( IBaseStream::Position position );
```

| <u>Win</u> | <u>PM</u> | <u>Motif</u> |
|------------|-----------|--------------|
| <i>Y</i> | <i>N</i> | <i>N</i> |

Inherited Public Functions

| IBaseStream | | |
|----------------|---------------------|-----------------------|
| context | logicalEndOfStream | seek |
| defaultContext | physicalEndOfStream | seekRelative |
| flush | position | setContext |
| isWriteable | read | setLogicalEndOfStream |

| IVBase | | |
|-------------|----------|--|
| asDebugInfo | asString | |

| IBase | | |
|-------------|-------------|----------------|
| asDebugInfo | messageFile | setMessageFile |
| asString | messageText | version |

Protected Functions

Buffer Management

Use these members to implement the reading and writing of data to a memory stream.

handleReadBufferEmpty

Handles reading of data from the stream.

```
virtual void
    handleReadBufferEmpty( void* target,
                          size_t byteCount );
```

| <u>Win</u> | <u>PM</u> | <u>Motif</u> |
|------------|-----------|--------------|
| <i>Y</i> | <i>N</i> | <i>N</i> |

The parameters are the following:

target The address to store the data that is read.
byteCount The number of bytes to be read.

IMemoryStream

handleWriteBufferFull

Handles writing of data to the stream.

```
virtual void  
    handleWriteBufferFull( const void* source,  
                           size_t byteCount );
```

| <u>Win</u> | <u>PM</u> | <u>Motif</u> |
|------------|-----------|--------------|
| <i>Y</i> | <i>N</i> | <i>N</i> |

The parameters are the following:

source The address of the data to write.
byteCount The number of bytes to be written.

Stream Positioning

These members control the positioning of a file stream.

setPhysicalEndOfStream

Sets the physical size of the stream to *position*.

```
IMemoryStream&  
    setPhysicalEndOfStream( IBaseStream::Position position );
```

| <u>Win</u> | <u>PM</u> | <u>Motif</u> |
|------------|-----------|--------------|
| <i>Y</i> | <i>N</i> | <i>N</i> |

Inherited Protected Functions

| IBaseStream | | |
|-----------------------|-----------------------|--|
| handleReadBufferEmpty | handleWriteBufferFull | |

Inherited Protected Data

| IBase | | |
|-------------|---------------|--|
| recoverable | unrecoverable | |



IMessageText

Derivation Inherits from none.

Inherited By None.

Header File
imsgtext.hpp

| Members | Member | Page | Member | Page |
|---------|-----------------------|------|----------------|------|
| | Constructor | 426 | setDefaultText | 428 |
| | operator = | 428 | text | 428 |
| | operator const char * | 428 | ~IMessageText | 428 |

The IMessageText class loads message text from a message file. When the User Interface Class 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. 394).

- 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

You can construct, destruct, copy, and assign objects of this class.

IMessageText

IMessageText

```
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 User Interface Class 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

A text string you insert into the message. Optional.

IMessageText

```
2 IMessageText( const IMessageText& text );
```

You can construct objects of this class using the User Interface Class 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 );
```

~IMessageText

```
~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;
```



IMetaType

Derivation Inherits from none.

Inherited By None.

Header File
imetatyp.hpp

| Members | Member | Page | Member | Page |
|---------|--------------|------|--------------|------|
| | Constructor | 429 | operator == | 429 |
| | name | 431 | operator >>= | 431 |
| | operator != | 429 | typeInfo | 431 |
| | operator <<= | 430 | ~IMetaType | 430 |
| | operator = | 430 | | |

The IMetaType class is used to stream type information. It wraps IMetaTypeInfo objects providing operators for streaming in and out.

Public Functions

Comparisons

Use these operators to compare the types of two objects.

operator != Compares two IMetaType objects for inequality. Returns true if this object and the argument object are different.

| | | | |
|---|------------|-----------|--------------|
| IBoollean | Win | PM | Motif |
| operator !=(const IMetaType& type) const; | <i>Y</i> | <i>N</i> | <i>N</i> |

operator == Compares two IMetaType objects for equality. Returns true if this object and the argument object are the same.

| | | | |
|---|------------|-----------|--------------|
| IBoollean | Win | PM | Motif |
| operator ==(const IMetaType& type) const; | <i>Y</i> | <i>N</i> | <i>N</i> |

Constructors

You can construct, copy, assign, and delete IMetaType objects.

IMetaType Constructs IMetaType objects.

IMetaType

| | | | | |
|----------|---------------------------|-------------------|------------------|---------------------|
| 1 | <code>IMetaType();</code> | <u>Win</u> | <u>PM</u> | <u>Motif</u> |
| | | <i>Y</i> | <i>N</i> | <i>N</i> |

Provides the default constructor.

| | | | | |
|----------|--|-------------------|------------------|---------------------|
| 2 | <code>IMetaType(const IMetaTypeInfo& typeInfo);</code> | <u>Win</u> | <u>PM</u> | <u>Motif</u> |
| | | <i>Y</i> | <i>N</i> | <i>N</i> |

You use this function to construct an IMetaType based on the given IMetaTypeInfo object.

| | | | | |
|----------|--|-------------------|------------------|---------------------|
| 3 | <code>IMetaType(const IMetaType& type);</code> | <u>Win</u> | <u>PM</u> | <u>Motif</u> |
| | | <i>Y</i> | <i>N</i> | <i>N</i> |

You use this function to construct an IMetaType object from another IMetatype object.

| | | | | |
|----------|---|-------------------|------------------|---------------------|
| 4 | <code>IMetaType(const char* theClassName);</code> | <u>Win</u> | <u>PM</u> | <u>Motif</u> |
| | | <i>Y</i> | <i>N</i> | <i>N</i> |

You use this function to construct an IMetaType from a string that represents the name of a class.

operator = An assignment operator called when an IMetaType object is assigned to another IMetaType object. It returns a non-const reference to the object on the left.

| | | | |
|--|-------------------|------------------|---------------------|
| <code>IMetaType& operator =(const IMetaType& type);</code> | <u>Win</u> | <u>PM</u> | <u>Motif</u> |
| | <i>Y</i> | <i>N</i> | <i>N</i> |

~IMetaType

| | | | |
|--|-------------------|------------------|---------------------|
| <code>virtual ~IMetaType();</code> | <u>Win</u> | <u>PM</u> | <u>Motif</u> |
| | <i>Y</i> | <i>N</i> | <i>N</i> |

The destructor.

Stream Operators

Use these operators to stream-in or stream-out the data of an object.

operator <<= The stream-in operator called to stream-in data. It returns a reference to the stream.

| | | | |
|--|-------------------|------------------|---------------------|
| <code>IBaseStream& operator <<=(IBaseStream& baseStream);</code> | <u>Win</u> | <u>PM</u> | <u>Motif</u> |
| | <i>Y</i> | <i>N</i> | <i>N</i> |

IMetaType

operator >>= The stream-out operator called to stream-out data. It returns a reference to the stream.

| | | | |
|--|-------------------|------------------|---------------------|
| IBaseStream& operator >>=(IBaseStream& baseStream) const; | <u>Win</u> | <u>PM</u> | <u>Motif</u> |
| | <i>Y</i> | <i>N</i> | <i>N</i> |

Type Information

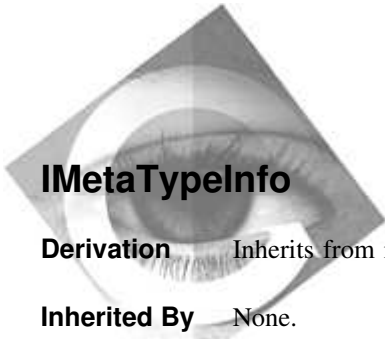
Use these members to determine the type information of an object.

name Provides the name of the class represented by this IMetaType object.

| | | | |
|---------------------------------|-------------------|------------------|---------------------|
| void name(IString&) const; | <u>Win</u> | <u>PM</u> | <u>Motif</u> |
| | <i>Y</i> | <i>N</i> | <i>N</i> |

typeInfo Provides the IMetaTypeInfo object associated with this object.

| | | | |
|---|-------------------|------------------|---------------------|
| const IMetaTypeInfo& typeInfo() const; | <u>Win</u> | <u>PM</u> | <u>Motif</u> |
| | <i>Y</i> | <i>N</i> | <i>N</i> |



IMetaTypeInfo

Derivation Inherits from none.

Inherited By None.

Header File
imetatyp.hpp

| Members | Member | Page |
|---------|-------------|------|
| | name | 433 |
| | operator != | 433 |
| | operator == | 433 |

The IMetaTypeInfo class provides type information for class objects.

To take advantage of type information, include a macro in both your declaration (.hpp) and implementation (.cpp) files, based on the type of classes that you want to use run-time type information (RTTI) for. Use the macro that corresponds to the category of class you are using as listed below, where:

- aname and _tclass represent the class name.
- _ptype, _ptypeK and _ptypeV represent the template class arguments.

The types of classes and their associated macros are as follows:

- Concrete classes
 - #define TypeExtensionDeclarationsMacro(aname)
 - #define TypeExtensionMacro(aname)
- Abstract classes (i.e., Classes with pure virtual functions)
 - #define TypeExtensionDeclarationsMacro_Abstract(aname)
 - #define TypeExtensionMacro_Abstract(aname)
- Concrete template classes with one template argument
 - #define TypeExtensionTemplateDeclarationsMacro(_tclass,_ptype)
 - #define TypeExtensionTemplateMacro(_tclass,_ptype)
- Abstract template classes with one template argument
 - #define TypeExtensionTemplateDeclarationsMacro_Abstract(_tclass,_ptype)
 - #define TypeExtensionTemplateMacro_Abstract(_tclass,_ptype)
- Concrete template classes with two template arguments

IMetaTypeInfo

- #define
TypeExtensionTemplatePairDeclarationsMacro(_tclass,_ptypeK,_ptypeV)
- #define TypeExtensionTemplatePairMacro(_tclass,_ptypeK,_ptypeV)
- Abstract template classes with two template arguments
 - #define
TypeExtensionTemplatePairDeclarationsMacro_Abstract(_tclass,_ptypeK,_ptypeV)
 - #define
TypeExtensionTemplatePairMacro_Abstract(_tclass,_ptypeK,_ptypeV)

In the future, this class's functionality will be provided by the compiler's native RTTI support.

Public Functions

Comparisons

Use these operators to compare the types of two objects.

operator != Compares two IMetaTypeInfo objects for inequality. It returns true if this object and the argument object are different.

| | | | | |
|---|--|------------|-----------|--------------|
| int | | <u>Win</u> | <u>PM</u> | <u>Motif</u> |
| operator !=(const IMetaTypeInfo& typeInfo) const; | | <i>Y</i> | <i>N</i> | <i>N</i> |

operator == Compares two IMetaTypeInfo objects for equality. It returns true if this object and the argument object are the same.

| | | | | |
|---|--|------------|-----------|--------------|
| int | | <u>Win</u> | <u>PM</u> | <u>Motif</u> |
| operator ==(const IMetaTypeInfo& typeInfo) const; | | <i>Y</i> | <i>N</i> | <i>N</i> |

Type Information

Use these members to determine the type information of an object.

name Provides the name of the class represented by this IMetaTypeInfo object.

| | | | | |
|---------------|--|------------|-----------|--------------|
| const char* | | <u>Win</u> | <u>PM</u> | <u>Motif</u> |
| name() const; | | <i>Y</i> | <i>N</i> | <i>N</i> |

IMetaTypeInfo

Associated Globals

copy AType* copy(const AType& sourceRef);

Performs a polymorphic copy. Returns a copy of the object that has the same type as the actual object referenced by the input parameter.

copyPointer AType* copyPointer(const AType* sourcePtr);

Performs a polymorphic copy. This function is the same as the copy global function except that the argument is a pointer instead of a reference. Returns a copy of the pointer to the object that has the same type as the actual object referenced by the input parameter.

createNewObject

void createNewObject(AType *& theResult, const IMetaType& theType);

Creates an object of the type represented by an IMetaType object. The pointer to the object is returned by the parameter theResult.

dynamicCastTo

void dynamicCastTo(AType *& target, BType* object);

Attempts to cast an object pointed to by the BType pointer to be an object pointed to by the AType pointer. If the object can be casted to AType, a non-null pointer is set for the target parameter. Otherwise, *target* is set to 0.



INotificationEvent

Derivation IBase
INotificationEvent

Inherited By None.

Header File
inotifev.hpp

| Members | Member | Page | Member | Page |
|---------|------------------------|------|------------------------|------|
| | Constructor | 435 | operator = | 436 |
| | eventData | 436 | setEventData | 437 |
| | hasNotifierAttrChanged | 436 | setNotifierAttrChanged | 437 |
| | notificationId | 436 | setObserverData | 437 |
| | notifier | 436 | ~INotificationEvent | 436 |
| | observerData | 437 | | |

The INotificationEvent class 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

You can construct, destruct, and assign objects of this class.

INotificationEvent

```
1 INotificationEvent(  
    const INotificationId& identifier,  
    INotifier& notifier,  
    Boolean notifierAttrChanged = true,  
    const IEventData& eventData = IEventData ( ),  
    const IEventData& observerData = IEventData ( ) );
```

INotificationEvent

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 );
```

~INotificationEvent

```
~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.

INotificationEvent

```
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 |

INotificationEvent

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 | 440 | notifyObservers | 441 |
| addObserver | 441 | observerList | 442 |
| disableNotification | 440 | removeAllObservers | 442 |
| enableNotification | 440 | removeObserver | 442 |
| isEnabledForNotification | 441 | ~INotifier | 440 |

The INotifier class 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 ways:

- 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.



See *Building VisualAge C++ Parts for Fun and Profit* for more information on part construction.

Public Functions

Constructors

You cannot construct objects of this class because it is an abstract base class.

INotifier

```
INotifier();
```

~INotifier

```
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 database be accessible before processing a retrieve function.

```
virtual INotifier&  
enableNotification( Boolean enable = true ) = 0;
```

INotifier

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.

INotifier

```
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 IObservableList&
    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 | |



Inherited By None.

Header File
iobsrvr.hpp

| Members | Member | Page | Member | Page |
|---------|---------------------------|------|------------------------------|------|
| | Constructor | 445 | stopHandlingNotificationsFor | 445 |
| | dispatchNotificationEvent | 446 | ~IObserver | 444 |
| | handleNotificationsFor | 445 | | |

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

Constructors

Only derived classes can create objects of this class. To enforce this, the only constructor has protected access.

~IObserver

```
virtual
~IObserver();
```

Event Dispatching

Use these members to evaluate events and determine if it is appropriate for an observer object to process them. These members also attach the observer to and detach the observer from the INotifier object.

IObserver

handleNotificationsFor

Attaches the observer to the INotifier object. 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 INotifier object.

```
virtual IObserver&
stopHandlingNotificationsFor( INotifier& notifier );
```

Inherited Public Functions

| IVBase | | |
|-------------|----------|--|
| asDebugInfo | asString | |

| IBase | | |
|-------------|-------------|----------------|
| asDebugInfo | messageFile | setMessageFile |
| asString | messageText | version |

Protected Functions

Constructors

Only derived classes can create objects of this class. To enforce this, the only constructor has protected access.

IObserver The default constructor.

```
IObserver();
```

Event Dispatching

Use these members to evaluate events and determine if it is appropriate for an observer object to process them. These members also attach the observer to and detach the observer from the INotifier object.

IObserver

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 | 447 | numberOfElements | 448 |
| | add | 448 | remove | 448 |
| | elementAt | 448 | removeAll | 448 |
| | isEmpty | 448 | removeAt | 448 |
| | notifyObservers | 449 | ~IObserverList | 447 |

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

Constructors

You can construct and destruct objects of this class.

IObserverList

```
IObserverList();
```

You can only construct objects of this class using the default constructor that takes no arguments.

~IObserverList

```
virtual  
~IObserverList();
```

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.

IObserverList

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 450)



IObserverList::Cursor

Derivation IBase
IVBase
IObserverList::Cursor

Inherited By None.

Header File
iobslist.hpp

| Members | Member | Page | Member | Page |
|---------|-------------|------|---------------|------|
| | Constructor | 450 | setToLast | 451 |
| | Cursor | 450 | setToNext | 451 |
| | invalidate | 450 | setToPrevious | 451 |
| | isValid | 451 | ~Cursor | 450 |
| | setToFirst | 451 | | |

The IObserverList::Cursor class is a nested cursor class used to iterate over the observers added to an INotifier.

Public Functions

Constructors

You can construct and destruct objects of this class.

Cursor Create an IObserverList::Cursor by providing a reference to an IObserverlist.

```
Cursor( IObserverList& observerList );
```

~Cursor

```
virtual  
~Cursor();
```

Cursor Movement

These members provide cursor movement operations.

invalidate Marks the cursor as invalid.

IObserverList::Cursor

```
virtual void  
    invalidate();
```

isValid Returns true if the cursor is on a valid observer.

```
virtual Boolean  
    isValid() const;
```

setToFirst Sets 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 |

IObserverList::Cursor

Inherited Protected Data

| IBase | | |
|--------------------|----------------------|--|
| recoverable | unrecoverable | |



IOutOfMemory

Derivation IException
 IResourceExhausted
 IOutOfMemory

Inherited By None.

Header File
 iexcbase.hpp

| Members | Member | Page |
|----------------|---------------|-------------|
| | Constructor | 453 |
| | name | 454 |

The IOutOfMemory class represents 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

Constructors

You can construct objects of this class.

IOutOfMemory

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. 403) to specify the severity of the error. The default is unrecoverable.

- Using the macros discussed in IException (p. 394). The User Interface Class Library provides these macros to make creating exceptions easier for you.

```
IOutOfMemory( const char* errorText,  
               unsigned long errorId,  
               Severity severity = IException::unrecoverable );
```

IOutOfMemory

Exception Type

Use these members to determine the name (type) of the exception. They are 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 | 455 |
| | name | 456 |

The IOutOfSystemResource class represents 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

Constructors

You can construct objects of this class.

IOutOfSystemResource

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. 403) to specify the severity of the error. The default is unrecoverable.

- Using the macros discussed in IException (p. 394). The User Interface Class Library provides these macros to make creating exceptions easier for you.

IOutOfSystemResource

```
IOutOfSystemResource(  
    const char* errorText,  
    unsigned long errorId,  
    Severity severity = IException::unrecoverable );
```

Exception Type

Use these members to determine the name (type) of the exception. They are 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 | 457 |
| | name | 458 |

The IOutOfWindowResource class represents 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

Constructors

You can construct objects of this class.

IOutOfWindowResource

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. 403) to specify the severity of the error. The default is unrecoverable.

- Using the macros discussed in IException (p. 394). The User Interface Class Library provides these macros to make creating exceptions easier for you.

IOutOfWindowResource

```
IOutOfWindowResource(  
    const char* errorText,  
    unsigned long errorId,  
    Severity severity = IException::unrecoverable );
```

Exception Type

Use these members to determine the name (type) of the exception. They are 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 | 460 | operator - | 461 | |
| asDebugInfo | 461 | operator -= | 463 | |
| asString | 461 | operator /= | 463 | |
| coord1 | 461 | operator < | 460 | |
| coord2 | 461 | operator <= | 460 | |
| distanceFrom | 464 | operator == | 460 | |
| dotProduct | 464 | operator > | 460 | |
| maximum | 464 | operator >= | 460 | |
| minimum | 464 | scaleBy | 463 | |
| operator != | 460 | scaledBy | 463 | |
| operator %= | 462 | setCoord1 | 461 | |
| operator *= | 462 | setCoord2 | 462 | |
| operator += | 463 | transpose | 464 | |

The IPair class provides generic ordered pairs of coordinates. The class serves as the base for the following specific ordered-pair classes:

- IPoint (p. 467)
- ISize (p. 500)
- IRange (p. 474)

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.

IPair

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.

IPair

```
 IPair( Coord init );
```

IPair

2 IPair();

3 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. 508) containing a diagnostic representation of the object.

```
IString  
asDebugInfo() const;
```

asString Converts the ordered pair (a, b) to an IString("(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

```
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 nonmember 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 parameters:

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*.

```
1 IPair&  
    operator %=( long divisor );
```

```
2 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 performs the product function between each coordinate and the *multiplier*.

```
1 IPair&  
    operator *=( double multiplier );
```

```
2 IPair&  
    operator *=( const IPair& aPair );
```

IPair

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*.

```
1 IPair&  
operator /=( const IPair& aPair );
```

```
2 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. 463), 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.

IPair

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).

Associated Globals

operator % `IPair operator %(const IPair& aPair1, long divisor);`

Returns an ordered pair whose coordinates are the remainder of the corresponding coordinates of the IPair (*aPair1*) and the *divisor*.

operator % `IPair operator %(const IPair& pair1, const IPair& pair2);`

Returns an ordered pair whose coordinates are the remainder of the corresponding coordinates of *pair1* and *pair2*.

operator * `IPair operator *(const IPair& pair1, double multiplier);`

Returns an ordered pair whose coordinates are the product of *pair1* and *multiplier*.

The library performs the product function between each coordinate of *pair1* and the *multiplier*.

operator * `IPair operator *(const IPair& pair1, const IPair& pair2);`

Returns an ordered pair whose coordinates are the product of *pair1* and *pair2*.

The library performs the product function between the corresponding coordinates: coord1 of *pair1* with coord1 of *pair2* and coord2 with coord2.

operator + `IPair operator +(const IPair& pair1, const IPair& pair2);`

IPair

Returns an ordered pair whose coordinates are the sums of the corresponding coordinates of *pair1* and *pair2*.

operator - IPair operator -(const IPair& pair1, const IPair& pair2);

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.

operator / IPair operator /(const IPair& pair1, const IPair& pair2);

Returns an ordered pair whose coordinates are the quotient of the corresponding coordinates of *pair1* and *pair2* specified parameter:

The library performs the quotient function between the corresponding coordinates: coord1 of *pair1* with coord1 of *pair2* and coord2 with coord2.

operator / IPair operator /(const IPair& pair1, double divisor);

Returns an ordered pair whose coordinates are the quotient of the corresponding coordinates of *pair1* and *divisor*

The library performs the quotient function between each coordinate of *pair1* and the *divisor*.

operator << ostream& operator <<(ostream& aStream, const IPair& aRectangle);

Writes ordered pairs to ostream in the conventional manner.

transpose IPair transpose(const IPair& aPair);

Swaps the coordinates of the ordered pair. The friend version of this function returns a new pair with transposed coordinates.



IPoint

Derivation IBase
 IPair
 IPoint

Inherited By None.

Header File
 ipoint.hpp

| Members | Member | Page | Member | Page |
|---------|-------------|------|--------|------|
| | Constructor | 467 | setY | 468 |
| | asPOINTL | 468 | x | 468 |
| | setX | 468 | y | 468 |

The IPoint class represent points in two-dimensional space. In addition to all the functions inherited from its base class, IPair (p. 459), 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.

IPoint

- 1** IPoint(const IPair& pair);
- 2** IPoint();
- 3** IPoint(Coord x,
 Coord y);

IPoint

| | | | | |
|----------|--------------------------------------|------------|-----------|--------------|
| 4 | IPoint(const struct _POINTL& pt1); | <u>Win</u> | <u>PM</u> | <u>Motif</u> |
| | | <i>Y</i> | <i>Y</i> | <i>N</i> |

| | | | | |
|----------|---------------------------------------|------------|-----------|--------------|
| 5 | IPoint(const struct tagPOINT& pt1); | <u>Win</u> | <u>PM</u> | <u>Motif</u> |
| | | <i>Y</i> | <i>N</i> | <i>N</i> |

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 | <u>Win</u> | <u>PM</u> | <u>Motif</u> |
| asPOINTL() const; | <i>Y</i> | <i>Y</i> | <i>N</i> |

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 | |



Inherited By None.

Header File
iptarray.hpp

| Members | Member | Page | Member | Page |
|---------|-------------|------|--------------|------|
| | Constructor | 471 | remove | 472 |
| | add | 471 | resize | 472 |
| | insert | 471 | reverse | 472 |
| | operator != | 470 | reversed | 472 |
| | operator = | 471 | size | 472 |
| | operator == | 470 | ~IPointArray | 471 |
| | operator [] | 472 | | |

The IPointArray class represents 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; | <u>Win</u> | <u>PM</u> | <u>Motif</u> |
| | Y | Y | N |

operator == Returns true if the arrays are the same length and have identical points.

| | | | |
|---|------------|-----------|--------------|
| Boolean | <u>Win</u> | <u>PM</u> | <u>Motif</u> |
| operator ==(const IPointArray& pointArray) const; | Y | Y | N |

Constructors

You can construct, copy, and assign objects of this class.

IPointArray

IPointArray Use this function to construct an IPointArray object.

| | | | | |
|----------|---|-------------------|------------------|---------------------|
| 1 | <code>IPointArray(unsigned long dimension = 0, const IPoint* array = 0);</code> | <u>Win</u> | <u>PM</u> | <u>Motif</u> |
| | | <i>Y</i> | <i>Y</i> | <i>N</i> |

Use this function to construct an 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 | <code>IPointArray(const IPointArray& pointArray);</code> | <u>Win</u> | <u>PM</u> | <u>Motif</u> |
| | | <i>Y</i> | <i>Y</i> | <i>N</i> |

Use this function to construct an IPointArray object from an existing IPointArray object.

operator = Use this function to assign one IPointArray object to another.

| | | | |
|--|-------------------|------------------|---------------------|
| <code>IPointArray& operator =(const IPointArray& pointArray);</code> | <u>Win</u> | <u>PM</u> | <u>Motif</u> |
| | <i>Y</i> | <i>Y</i> | <i>N</i> |

Use this function to assign one IPointArray object to another. The target IPointArray object is increased or decreased to the size of the source IPointArray object.

~IPointArray

| | | | |
|------------------------------|-------------------|------------------|---------------------|
| <code>~IPointArray();</code> | <u>Win</u> | <u>PM</u> | <u>Motif</u> |
| | <i>Y</i> | <i>Y</i> | <i>N</i> |

Data Access

Use these members to access attributes of objects of this class.

add Adds a point to the end of the array.

| | | | |
|---|-------------------|------------------|---------------------|
| <code>IPointArray& add(const IPoint& point);</code> | <u>Win</u> | <u>PM</u> | <u>Motif</u> |
| | <i>Y</i> | <i>Y</i> | <i>N</i> |

insert Inserts a point before the index specified.

IPointArray

| | | | |
|--|------------------------|-----------------------|--------------------------|
| IPointArray& insert(unsigned long index, const IPoint& point); | <u>Win</u> <i>Y</i> | <u>PM</u> <i>Y</i> | <u>Motif</u> <i>N</i> |
|--|------------------------|-----------------------|--------------------------|

operator []

Returns a reference to the point at the specified index.

| | | | |
|--|------------------------|-----------------------|--------------------------|
| 1 IPoint& operator [] (unsigned long index); | <u>Win</u> <i>Y</i> | <u>PM</u> <i>Y</i> | <u>Motif</u> <i>N</i> |
|--|------------------------|-----------------------|--------------------------|

| | | | |
|--|------------------------|-----------------------|--------------------------|
| 2 const IPoint& operator [] (unsigned long index) const; | <u>Win</u> <i>Y</i> | <u>PM</u> <i>Y</i> | <u>Motif</u> <i>N</i> |
|--|------------------------|-----------------------|--------------------------|

remove Removes a point at the specified index.

| | | | |
|--|------------------------|-----------------------|--------------------------|
| IPointArray& remove(unsigned long index); | <u>Win</u> <i>Y</i> | <u>PM</u> <i>Y</i> | <u>Motif</u> <i>N</i> |
|--|------------------------|-----------------------|--------------------------|

resize Increases or decreases the size of the array. New points are initialized to 0,0.

| | | | |
|--|------------------------|-----------------------|--------------------------|
| IPointArray& resize(unsigned long newsize); | <u>Win</u> <i>Y</i> | <u>PM</u> <i>Y</i> | <u>Motif</u> <i>N</i> |
|--|------------------------|-----------------------|--------------------------|

reverse Reverses the elements in the array.

| | | | |
|----------------------------|------------------------|-----------------------|--------------------------|
| IPointArray& reverse(); | <u>Win</u> <i>Y</i> | <u>PM</u> <i>Y</i> | <u>Motif</u> <i>N</i> |
|----------------------------|------------------------|-----------------------|--------------------------|

reversed Returns a copy of the point array with its elements reversed.

| | | | |
|----------------------------------|------------------------|-----------------------|--------------------------|
| IPointArray reversed() const; | <u>Win</u> <i>Y</i> | <u>PM</u> <i>Y</i> | <u>Motif</u> <i>N</i> |
|----------------------------------|------------------------|-----------------------|--------------------------|

size Returns the dimension of the array.

| | | | |
|--------------------------------|------------------------|-----------------------|--------------------------|
| unsigned long size() const; | <u>Win</u> <i>Y</i> | <u>PM</u> <i>Y</i> | <u>Motif</u> <i>N</i> |
|--------------------------------|------------------------|-----------------------|--------------------------|

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 | 474 | setLowerBound | 475 |
| | | includes | 475 | setUpperBound | 475 |
| | | lowerBound | 475 | upperBound | 475 |

The IRange class represents 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.

IRange

- 1 IRange(Coord lower,
Coord upper);
- 2 IRange();
- 3 IRange(const IPair& aPair);

Coordinates

Use these members to query and change the ordered pair of integers in an IRange object.

IRange

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 |

IRange

| IPair | | |
|---------|-------------|-----------|
| 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 | 479 | | minXMinY | 487 |
| | area | 481 | | minY | 481 |
| | asDebugInfo | 480 | | moveBy | 483 |
| | asRECTL | 480 | | movedBy | 483 |
| | asString | 481 | | movedTo | 483 |
| | bottom | 488 | | moveTo | 483 |
| | bottomCenter | 488 | | operator != | 478 |
| | bottomLeft | 488 | | operator & | 486 |
| | bottomRight | 488 | | operator &= | 486 |
| | center | 489 | | operator == | 479 |
| | centerAt | 482 | | operator | 486 |
| | centeredAt | 482 | | operator = | 486 |
| | centerXCenterY | 486 | | right | 489 |
| | centerXMaxY | 486 | | rightCenter | 489 |
| | centerXMinY | 487 | | scaleBy | 483 |
| | contains | 490 | | scaledBy | 483 |
| | expandBy | 482 | | shrinkBy | 484 |
| | expandedBy | 482 | | shrunkBy | 484 |
| | height | 481 | | size | 482 |
| | intersects | 490 | | sizeBy | 484 |
| | left | 489 | | sizedBy | 485 |
| | leftCenter | 489 | | sizedTo | 485 |
| | maxX | 481 | | sizeTo | 485 |
| | maxXCenterY | 487 | | top | 489 |
| | maxXMaxY | 487 | | topCenter | 489 |
| | maxXMinY | 487 | | topLeft | 490 |
| | maxY | 481 | | topRight | 490 |
| | minX | 481 | | validate | 491 |
| | minXCenterY | 487 | | width | 482 |
| | minXMaxY | 487 | | | |

The IRectangle class represents 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

IRectangle objects are 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 top-left corner 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 bottom-right corner. Conversely, in a coordinate space where 0,0 is the bottom-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 bottom left, and the maximum corner the top right.

IRectangle provides some member functions that 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, and 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 through 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.



You can also construct objects of this class by providing a Presentation Manager Toolkit RECTL structure.



You can also construct objects of this class by providing Windows toolkit RECT or RECTL structures.

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;
```

IRectangle

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 User Interface Class 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.

IRectangle

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.

IRectangle

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& rect1); | <u>Win</u> <i>Y</i> | <u>PM</u> <i>Y</i> | <u>Motif</u> <i>N</i> |
|----------|---|-------------------------------|------------------------------|---------------------------------|

Constructs a rectangle from a PM toolkit RECTL structure.

| | | | | |
|----------|---|-------------------------------|------------------------------|---------------------------------|
| 7 | IRectangle(const struct tagRECT& rect); | <u>Win</u> <i>Y</i> | <u>PM</u> <i>N</i> | <u>Motif</u> <i>N</i> |
|----------|---|-------------------------------|------------------------------|---------------------------------|

Constructs a rectangle from a Windows toolkit RECT structure.

| | |
|----------|---|
| 8 | IRectangle(Coord width, Coord height); |
|----------|---|

Constructs a rectangle of the specified size. The size is specified as follows:

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.

| | | | |
|---|-------------------------------|------------------------------|---------------------------------|
| <pre>struct _RECTL asRECTL() const;</pre> | <u>Win</u> <i>Y</i> | <u>PM</u> <i>Y</i> | <u>Motif</u> <i>N</i> |
|---|-------------------------------|------------------------------|---------------------------------|

IRectangle



Renders the rectangle as a Presentation Manager Toolkit RECTL structure.



Renders the rectangle as a Windows 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 through the origin.

```
Coord  
    minX() const;
```

minY Returns the Y-coordinate of the horizontal line that passes through the origin of the rectangle.

IRectangle

| | |
|--------------|--|
| | <pre>Coord minY() const;</pre> |
| size | Returns the ISize(width, height). |
| | <pre>ISize size() const;</pre> |
| 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. |
| | <pre>Coord width() const;</pre> |

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. |
| | <pre>IRectangle& centerAt(const IPoint& point);</pre> |
| centeredAt | Same as centerAt , but returns a new rectangle, leaving the original unmodified. |
| | <pre>IRectangle centeredAt(const IPoint& point) const;</pre> |
| 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. |
| | <pre>1 IRectangle& expandBy(const IPair& pair);</pre> |
| | <pre>2 IRectangle& expandBy(Coord coord);</pre> |
| expandedBy | Same as expandBy , but returns a new rectangle, leaving the original unmodified. |
| | <pre>1 IRectangle expandedBy(const IPair& pair) const;</pre> |
| | <pre>2 IRectangle expandedBy(Coord coord) const;</pre> |

IRectangle

moveBy Moves the rectangle by the amount specified by *aPair*.

```
IRectangle&  
    moveBy( const IPair& pair );
```

movedBy Same as **moveBy**, but returns a new rectangle, leaving the original unmodified.

```
IRectangle  
    movedBy( const IPair& pair ) const;
```

movedTo Same as **moveTo**, 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-axis 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 **scaleBy**, but returns a new rectangle, leaving the original unmodified.

IRectangle

1 IRectangle
scaledBy(double xfactor,
double yfactor) const;

2 IRectangle
scaledBy(const IPair& pair) const;

3 IRectangle
scaledBy(Coord coord) const;

4 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.

1 IRectangle&
shrinkBy(Coord coord);

2 IRectangle&
shrinkBy(const IPair& pair);

shrunkBy

Same as **shrinkBy**, but returns a new rectangle, leaving the original unmodified.

1 IRectangle
shrunkBy(const IPair& pair) const;

2 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.

IRectangle

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 **sizeBy**, but returns a new rectangle, leaving the original unmodified.

1 IRectangle
sizedBy(double factor) const;

2 IRectangle
sizedBy(const IPair& pair) const;

3 IRectangle
sizedBy(Coord factor) const;

4 IRectangle
sizedBy(double xfactor,
double yfactor) const;

sizedTo Same as **sizeTo**, 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);

IRectangle

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.

IRectangle

```
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 through 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 through the origin.

```
IPoint  
    minXMaxY() const;
```

minXMinY Returns the X- and Y-coordinates of the origin corner of the rectangle.

```
IPoint  
    minXMinY() const;
```

IRectangle

Synonyms

Use these members when you are working in a coordinate space where 0,0 is the bottom-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 minY (p. 481).

```
Coord  
    bottom() const;
```

bottomCenter

Returns the X- and Y-coordinates of the bottom-center point of the rectangle. This is an alias for centerXMinY (p. 487).

```
IPoint  
    bottomCenter() const;
```

bottomLeft Returns the X- and Y-coordinates of the bottom-left corner of the rectangle This is an alias for minXMinY (p. 487).

```
IPoint  
    bottomLeft() const;
```

bottomRight Returns the X- and Y-coordinates of the bottom-right corner of the rectangle. This is an alias for maxXMinY (p. 487).

IRectangle

```
IPoint  
    bottomRight() const;
```

center Returns the X- and Y-coordinates of the center point of the rectangle. This is an alias for centerXCenterY (p. 486).

```
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 minX (p. 481).

```
Coord  
    left() const;
```

leftCenter Returns the X- and Y-coordinates of the left-center point of the rectangle. This is an alias for minXCenterY (p. 487).

```
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 maxX (p. 481).

```
Coord  
    right() const;
```

rightCenter Returns the X- and Y-coordinates of the right-center point of the rectangle. This is an alias for maxXCenterY (p. 487).

```
IPoint  
    rightCenter() const;
```

top Returns the Y-coordinate of the horizontal line that forms the top of the rectangle. This is an alias for maxY (p. 481).

```
Coord  
    top() const;
```

topCenter Returns the X- and Y-coordinates of the top-center point of the rectangle. This is an alias for centerXMaxY (p. 486).

```
IPoint  
    topCenter() const;
```

IRectangle

topLeft Returns the X- and Y-coordinates of the top-left corner of the rectangle. This is an alias for minXMaxY (p. 487).

```
IPoint  
    topLeft() const;
```

topRight Returns the X- and Y-coordinates of the top-right corner of the rectangle. This is an alias for maxXMaxY (p. 487).

```
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.

```
1 Boolean  
    contains( const IPoint& point ) const;
```

```
2 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.

Note: If the rectangle is invalid, the points for both the origin and the corner of the rectangle are reset to IPoint(0, 0).

```
IRectangle&
    validate();
```

Inherited Protected Data

| IBase | | |
|-------------|---------------|--|
| recoverable | unrecoverable | |

Nested Type Definitions

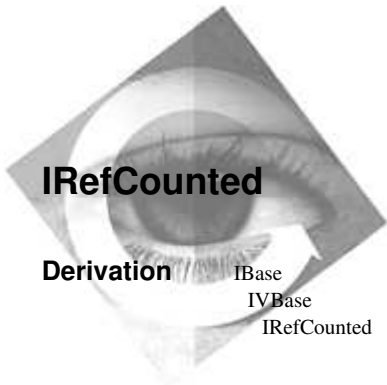
Coord typedef IPair::Coord Coord;

Type of the coordinate values; this must match the type of the coordinates supported by the IPair class.

Associated Globals

operator << ostream& operator <<(ostream& stream, const IRectangle& rectangle);

Writes rectangles to ostreams in the conventional manner.



Inherited By IDMItem IThreadFn
IDMOperation ITimerFn
IStringGeneratorFn

Header File
irefcnt.hpp

| Members | Member | Page | Member | Page |
|---------|-------------|------|--------------|------|
| | Constructor | 493 | useCount | 493 |
| | addRef | 492 | ~IRefCounted | 493 |
| | removeRef | 493 | | |

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. 493) 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. 495), 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.

IRefCounted

```
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

Constructors

These members are protected.

IRefCounted

```
IRefCounted();
```

~IRefCounted

```
~IRefCounted();
```

IRefCounted

Inherited Protected Data

| IBase | | |
|-------------|---------------|--|
| recoverable | unrecoverable | |



IReference

Derivation IBase
IReference

Inherited By None.

Header File
irefcnt.hpp

| Members | | Member | Page | Member | Page |
|----------------|--|---------------|-------------|---------------|-------------|
| | | Constructor | 496 | operator = | 496 |
| | | operator * | 497 | operator T * | 497 |
| | | operator -> | 497 | ~IReference | 497 |

The IReference class is a template class 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. 492). It must have the member functions IRefCounted::addRef (p. 492) and IRefCounted::removeRef (p. 493) 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.

Note:

1. The semantics of such reference or referent classes can have subtle complexities. The reference or the referent might behave in an extraordinary fashion.

IReference

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 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

You can construct, destruct, copy, and assign objects of this class.

IReference

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.

1 IReference < T >&
operator =(const IReference < T >& source);

2 IReference < T >&
operator =(T* p);

IReference

~IReference

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 | 498 |
| | name | 499 |

The IResourceExhausted class represents 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 the following:

- IOutOfMemory (p. 453)
- IOutOfSystemResource (p. 455)
- IOutOfWindowResource (p. 457)

Public Functions

Constructors

You can construct objects of this class.

IResourceExhausted

You can create objects of this class by doing the following:

- Using the constructor.
errorText The text describing this particular error.

IResourceExhausted

errorId The identifier you want to associate with this particular error.

severity Use the enumeration IException::Severity (p. 403) to specify the severity of the error. The default is unrecoverable.

- Using the macros discussed in IException (p. 394). The User Interface Class 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. They are used for logging out an exception object's error information.

name Returns the name of the object's class.

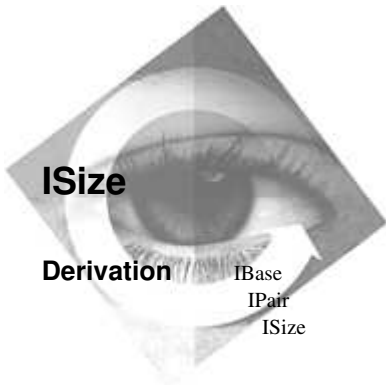
```
virtual const char*  
    name() const;
```

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 |



Inherited By None.

Header File
 ipoint.hpp

| Members | Member | Page | Member | Page |
|---------|-------------|------|-----------|------|
| | Constructor | 500 | setHeight | 501 |
| | asSIZEL | 501 | setWidth | 501 |
| | height | 501 | width | 501 |

The ISize class uses its coordinates to represent a rectangular size, in horizontal and vertical dimensions.

- PM

You can also construct objects of this class using the following:

 - 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

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 ISize objects.

ISize

- 1

ISize(const IPair& pair);
- 2

ISize();

ISize

```
3 ISize( Coord width,  
        Coord height );  
  
4 ISize( const SIZEL& sizl );  
  
5 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;
```

ISize

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 | 504 | notifyObservers | 505 |
| | addObserver | 506 | observerList | 506 |
| | deleteId | 507 | operator = | 504 |
| | disableNotification | 504 | removeAllObservers | 506 |
| | enableNotification | 505 | removeObserver | 506 |
| | isEnabledForNotification | 505 | ~IStandardNotifier | 504 |

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.

IStandardNotifier

Public Functions

Constructors

You can construct, destruct, assign, and copy objects of this class.

IStandardNotifier

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 );
```

~IStandardNotifier

```
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();
```

IStandardNotifier

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. 435) to call it. In this case, the construction of the INotificationEvent (p. 435) 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 |

ISandardNotifier

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 ISandardNotifier&
    addObserver( IObserver& observer,
                 const IEventData& userData = IEventData ( 0 ) );
```

observerList Returns the list of IObservers. The list is created if it does not exist.

```
virtual IObservableList&
    observerList() const;
```

removeAllObservers

Removes all observers from the object's observer list.

```
virtual ISandardNotifier&
    removeAllObservers();
```

removeObserver

Removes an observer from the object's observer list.

```
virtual ISandardNotifier&
    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. 435) to call it. In this case, the construction of the INotificationEvent (p. 435) object occurs in the code of the protected notifyObservers function.

```
virtual ISandardNotifier&
    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 IStandardNotifier 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 not, therefore, cast the pointer to the notifier data, but 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 | 513 | initBuffer | 550 |
| applyBitOp | 547 | insert | 518 |
| asDebugInfo | 515 | isAbbrevFor | 552 |
| asDouble | 543 | isAbbreviationFor | 535 |
| asInt | 543 | isAlphabetic | 541 |
| asLongLong | 544 | isAlphanumeric | 541 |
| asString | 515 | isASCII | 541 |
| asUnsigned | 544 | isBinaryDigits | 542 |
| asUnsignedLongLong | 544 | isControl | 542 |
| b2c | 511 | isDBCS | 533 |
| b2d | 511 | isDigits | 542 |
| b2x | 511 | isGraphics | 542 |
| buffer | 551 | isHexDigits | 542 |
| c2b | 512 | isInternationalized | 533 |
| c2d | 512 | isLike | 536 |
| c2x | 512 | isLowerCase | 542 |
| center | 516 | isMBCS | 533 |
| change | 516 | isPrintable | 542 |
| charType | 536 | isPunctuation | 543 |
| copy | 518 | isSBCS | 533 |
| d2b | 534 | isUpperCase | 543 |
| d2c | 534 | isValidDBCS | 534 |
| d2x | 535 | isValidMBCS | 534 |
| data | 551 | isWhiteSpace | 543 |
| defaultBuffer | 551 | lastIndexOf | 539 |
| disableInternationalization | 532 | lastIndexOfAnyBut | 539 |
| enableInternationalization | 532 | lastIndexOfAnyOf | 540 |
| findPhrase | 549 | leftJustify | 518 |
| includes | 535 | length | 537 |
| includesDBCS | 533 | lengthOf | 551 |
| includesMBCS | 533 | lengthOfWord | 545 |
| includesSBCS | 533 | lineFrom | 541 |
| indexOf | 525 | lowerCase | 519 |
| indexOfAnyBut | 526 | maxLong | 553 |
| indexOfAnyOf | 527 | null | 553 |
| indexOfPhrase | 545 | nullBuf | 551 |
| indexOfWord | 545 | numWords | 545 |

IString

| Member | Page | Member | Page |
|--------------------------|------|---------------------|------|
| occurrencesOf | 527 | setBuffer | 551 |
| operator & | 529 | size | 538 |
| operator &= | 529 | space | 546 |
| operator + | 529 | strip | 521 |
| operator += | 530 | stripBlanks | 522 |
| operator = | 530 | stripLeading | 522 |
| operator char * | 544 | stripLeadingBlanks | 523 |
| operator signed char * | 544 | stripTrailing | 523 |
| operator unsigned char * | 544 | stripTrailingBlanks | 524 |
| operator [] | 537 | subString | 538 |
| operator ^ | 530 | translate | 524 |
| operator ^= | 531 | upperCase | 525 |
| operator | 531 | word | 546 |
| operator = | 532 | wordIndexOfPhrase | 546 |
| operator ~ | 532 | words | 547 |
| overlayWith | 519 | x2b | 528 |
| remove | 520 | x2c | 528 |
| removeWords | 545 | x2d | 528 |
| reverse | 520 | zero | 553 |
| rightJustify | 520 | ~IString | 515 |

The IString class provides arrays of characters. These objects are functionally equivalent to objects of the class IOString (p. 287) with one major distinction: IStrings are indexed starting at 1 instead of 0. (Note that if you use an index of zero as a starting position in an IString member function, this index value is automatically converted to an index of 1.)

IString provides an operator char*. To access the actual string contained in an object of type IString, cast the assignment variable implicitly or explicitly.

IString member functions 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:

IString

- 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 for the byte index of a string and the last-byte index of string
- Word manipulation, such as the index of a word and the search for a word phrase.
- Support for mixed strings that contain both single-byte character set (SBCS) characters and double-byte character set (DBCS) characters.

When a program using IString objects 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 IString objects, such as `subString` (p. 538), `remove` (p. 520), and `translate` (p. 524), 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 IString objects 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 `n`th 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`.

IString objects are held in IBuffer objects, 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.



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.

1 IString&
b2c();

2 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.

1 static IString
b2d(const IString& aString);

2 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.

1 static IString
b2x(const IString& aString);

2 IString&
b2x();

Character Conversions

These members 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:

IString

```
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.

1 IString&
c2b();

2 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.

1 static IString
c2d(const IString& aString);

2 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.

1 static IString
c2x(const IString& aString);

2 IString&
c2x();

Constructors

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.

IString

- 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 pad 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 User Interface Class 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.

IString

```
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 anInt );
```

Construct a string with the ASCII representation of an integer value.

```
5  IString( unsigned anUnsigned );
```

Construct a string with the ASCII representation of an unsigned numeric value.

```
6  IString( long aLong );
```

Construct a string with the ASCII representation of a long numeric value.

IString

7 IString(unsigned long anUnsignedLong);

Construct a string with the ASCII representation of an unsigned long numeric value.

8 IString(long long aLongLong);

9 IString(unsigned long long anUnsignedLongLong);

10 IString(short aShort);

Construct a string with the ASCII representation of a short numeric value.

11 IString(unsigned short anUnsignedShort);

Construct a string with the ASCII representation of an unsigned short numeric value.

12 IString(double aDouble);

Construct a string with the ASCII representation of a double numeric value.

13 IString(char aChar);

Construct a string with a copy of the character. The string length is set to 1.

14 IString(unsigned char anUnsignedChar);

Construct a string with a copy of the unsigned character. The string length is set to 1.

15 IString(signed char aSignedChar);

Construct a string with a copy of the signed character. The string length is set to 1.

16 IString(const char* pChar);

Construct a string with a copy of the specified ASCIIZ string.

17 IString(const unsigned char* pUnsignedChar);

Construct a string with a copy of the specified unsigned ASCIIZ string.

18 IString(const signed char* pSignedChar);

Construct a string with a copy of the specified signed ASCIIZ string.

IString

```
19 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*).

```
20 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*).

~IString

```
~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. 308) 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*.

IString

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.

```
1 IString&  
  center( unsigned length,  
          char padCharacter = ' ' );
```

```
2 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 a 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

- 1** static IString
 change(const IString& aString,
 const char* pInputString,
 const char* pOutputString,
 unsigned startPos = 1,
 unsigned numChanges = (unsigned) UINT_MAX);

- 2** IString&
 change(const IString& inputString,
 const IString& outputString,
 unsigned startPos = 1,
 unsigned numChanges = (unsigned) UINT_MAX);

- 3** IString&
 change(const IString& inputString,
 const char* pOutputString,
 unsigned startPos = 1,
 unsigned numChanges = (unsigned) UINT_MAX);

- 4** IString&
 change(const char* pInputString,
 const IString& outputString,
 unsigned startPos = 1,
 unsigned numChanges = (unsigned) UINT_MAX);

- 5** IString&
 change(const char* pInputString,
 const char* pOutputString,
 unsigned startPos = 1,
 unsigned numChanges = (unsigned) UINT_MAX);

- 6** static IString
 change(const IString& aString,
 const IString& inputString,
 const IString& outputString,
 unsigned startPos = 1,
 unsigned numChanges = (unsigned) UINT_MAX);

- 7** static IString
 change(const IString& aString,
 const IString& inputString,
 const char* pOutputString,
 unsigned startPos = 1,
 unsigned numChanges = (unsigned) UINT_MAX);

IString

```
3 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.

```
1 static IString  
  copy( const IString& aString,  
        unsigned numCopies );
```

```
2 IString&  
  copy( unsigned numCopies );
```

insert Inserts the specified string after the specified location.

```
1 static IString  
  insert( const IString& aString,  
          const char* pInsert,  
          unsigned index = 0,  
          char padCharacter = ' ' );
```

```
2 IString&  
  insert( const IString& aString,  
          unsigned index = 0,  
          char padCharacter = ' ' );
```

```
3 IString&  
  insert( const char* pString,  
          unsigned index = 0,  
          char padCharacter = ' ' );
```

```
4 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.

IString

1 static IString
leftJustify(const IString& aString,
 unsigned length,
 char padCharacter = ' ');

2 IString&
leftJustify(unsigned length,
 char padCharacter = ' ');

lowerCase Translates all uppercase letters in the receiver to lowercase.

1 static IString
lowerCase(const IString& aString);

2 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*).

1 static IString
overlayWith(const IString& aString,
 const IString& anOverlay,
 unsigned index = 1,
 char padCharacter = ' ');

2 IString&
overlayWith(const IString& aString,
 unsigned index = 1,
 char padCharacter = ' ');

3 IString&
overlayWith(const char* pString,
 unsigned index = 1,
 char padCharacter = ' ');

IString

```
4 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.

```
1 IString&  
  remove( unsigned startPos,  
          unsigned numChars );
```

```
2 IString&  
  remove( unsigned startPos );
```

```
3 static IString  
  remove( const IString& aString,  
          unsigned startPos );
```

```
4 static IString  
  remove( const IString& aString,  
          unsigned startPos,  
          unsigned numChars );
```

reverse Reverses the receiver's contents.

```
1 IString&  
  reverse();
```

```
2 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.

IString

1 static IString
 rightJustify(const IString& aString,
 unsigned length,
 char padCharacter = ' ');

2 IString&
 rightJustify(unsigned length,
 char padCharacter = ' ');

strip

Strips both the 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. 508) object
- An IStringTest (p. 573) object

The default is white space.

1 IString&
 strip(const char* pString);

2 IString&
 strip();

3 IString&
 strip(char aCharacter);

4 IString&
 strip(const IString& aString);

5 IString&
 strip(const IStringTest& aTest);

6 static IString
 strip(const IString& aString,
 char aChar);

7 static IString
 strip(const IString& aString,
 const IString& aStringOfChars);

IString

```
8 static IString  
   strip( const IString& aString,  
          const char* pStringOfChars );
```

```
9 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. 521), which has been renamed to avoid a duplicate definition.

```
static IString  
   stripBlanks( const IString& aString );
```

stripLeading Strips the leading character or characters.

```
1 static IString  
   stripLeading( const IString& aString,  
                char aChar );
```

```
2 IString&  
   stripLeading();
```

```
3 IString&  
   stripLeading( char aCharacter );
```

```
4 IString&  
   stripLeading( const IString& aString );
```

```
5 IString&  
   stripLeading( const char* pString );
```

```
6 IString&  
   stripLeading( const IStringTest& aTest );
```

```
7 static IString  
   stripLeading( const IString& aString,  
                const IString& aStringOfChars );
```


IString

```
8 static IString  
   stripLeading( const IString& aString,  
                const char* pStringOfChars );
```

```
9 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. 522), which has been renamed to avoid a duplicate definition.

```
static IString  
   stripLeadingBlanks( const IString& aString );
```

stripTrailing Strips the trailing character or characters.

```
1 static IString  
   stripTrailing( const IString& aString,  
                  const IStringTest& aTest );
```

```
2 IString&  
   stripTrailing();
```

```
3 IString&  
   stripTrailing( char aCharacter );
```

```
4 IString&  
   stripTrailing( const IString& aString );
```

```
5 IString&  
   stripTrailing( const char* pString );
```

```
6 IString&  
   stripTrailing( const IStringTest& aTest );
```

IString

- 7** static IString
 stripTrailing(const IString& aString,
 char aChar);
- 8** static IString
 stripTrailing(const IString& aString,
 const IString& aStringOfChars);
- 9** 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. 523), 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.

- 1** static IString
 translate(const IString& aString,
 const char* pInputChars,
 const IString& outputChars,
 char padCharacter = ' ');
- 2** IString&
 translate(const IString& inputChars,
 const IString& outputChars,
 char padCharacter = ' ');
- 3** IString&
 translate(const IString& inputChars,
 const char* pOutputChars,
 char padCharacter = ' ');

IString

```
4 IString&
  translate( const char* pInputChars,
             const IString& outputChars,
             char padCharacter = ' ' );
```

```
5 IString&
  translate( const char* pInputChars,
             const char* pOutputChars,
             char padCharacter = ' ' );
```

```
6 static IString
  translate( const IString& aString,
             const IString& inputChars,
             const IString& outputChars,
             char padCharacter = ' ' );
```

```
7 static IString
  translate( const IString& aString,
             const IString& inputChars,
             const char* pOutputChars,
             char padCharacter = ' ' );
```

```
8 static IString
  translate( const IString& aString,
             const char* pInputChars,
             const char* pOutputChars,
             char padCharacter = ' ' );
```

upperCase Translates all lowercase letters in the receiver to uppercase.

```
1 IString&
  upperCase();
```

```
2 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. 573).

IString

- 1** unsigned
 indexOf(const IString& aString,
 unsigned startPos = 1) const;
- 2** unsigned
 indexOf(const char* pString,
 unsigned startPos = 1) const;
- 3** unsigned
 indexOf(char aCharacter,
 unsigned startPos = 1) const;
- 4** 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. 573) object.

- 1** unsigned
 indexOfAnyBut(const IString& validChars,
 unsigned startPos = 1) const;
- 2** unsigned
 indexOfAnyBut(const char* pValidChars,
 unsigned startPos = 1) const;
- 3** unsigned
 indexOfAnyBut(char validChar,
 unsigned startPos = 1) const;
- 4** 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. 573) object.

- 1** unsigned
 indexOfAnyOf(const char* pSearchChars,
 unsigned startPos = 1) const;
- 2** unsigned
 indexOfAnyOf(const IString& searchChars,
 unsigned startPos = 1) const;
- 3** unsigned
 indexOfAnyOf(char searchChar,
 unsigned startPos = 1) const;
- 4** 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. 525).

- 1** unsigned
 occurrencesOf(const IStringTest& aTest,
 unsigned startPos = 1) const;
- 2** unsigned
 occurrencesOf(const IString& aString,
 unsigned startPos = 1) const;
- 3** unsigned
 occurrencesOf(const char* pString,
 unsigned startPos = 1) const;

IString

```
4 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`.

```
1 IString&
  x2b();
```

```
2 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.

```
1 static IString
  x2c( const IString& aString );
```

```
2 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`.

```
1 static IString
  x2d( const IString& aString );
```

```
2 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.

1 IString
operator &(const char* pString) const;

2 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.

1 IString&
operator &=(const char* pString);

2 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.

IString

pString + aString

The first operand is a NULL-terminated character string and the second is an IString.

1 IString
operator +(const IString& aString) const;

2 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.

1 IString&
operator +=(const char* pString);

2 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

1 IString
operator ^(const char* pString) const;

2 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.

1 IString&
operator ^=(const IString& aString);

2 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.

1 IString
operator |(const char* pString) const;

2 IString
operator |(const IString& aString) const;

IString

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.

```
1 IString&  
   operator |=( const IString& aString );
```

```
2 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).

disableInternationalization

Disables locale-based string operations.

```
static void  
disableInternationalization();
```

enableInternationalization

Enables locale-based string operations.

```
static void  
enableInternationalization( Boolean enable = true );
```

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;
```

isInternationalized

Determines if locale-based string operation is in effect.

```
static Boolean  
isInternationalized();
```

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.

IString

```
Boolean  
    isSBCS() const;
```

isValidDBCS If no DBCS characters have a second byte of 0, true is returned.

Note: This function is interchangeable with isValidMBCS.

```
Boolean  
    isValidDBCS() const;
```

isValidMBCS If no MBCS characters have a second byte of 0, 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.

```
1 IString&  
    d2b();
```

```
2 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.

```
1 static IString  
    d2c( const IString& aString );
```

IString

| | | |
|------------|----------|--|
| | 2 | 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. |
| | 1 | static IString d2x(const IString& aString); |
| | 2 | 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.

- | | |
|----------|--|
| 1 | Boolean includes(const IStringTest& aTest) const; |
| 2 | Boolean includes(const IString& aString) const; |
| 3 | Boolean includes(const char* pString) const; |
| 4 | 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.

IString

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.

1 Boolean
isAbbreviationFor(const char* pFullString,
 unsigned minAbbrevLength = 0) const;

2 Boolean
isAbbreviationFor(const IString& fullString,
 unsigned minAbbrevLength = 0) const;

isLike

If the receiver matches the specified pattern, which can contain global (or wildcard) characters, true is returned.

- You can use the first global character to specify that 0 or more arbitrary characters are accepted. The default global 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 global character to specify that a single arbitrary character is accepted. The default global 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
```

1 Boolean
isLike(const char* pPattern,
 char zeroOrMore = ' * ',
 char anyChar = '?') const;

2 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 char&
operator [] (unsigned long index);
- 2 char&
operator [] (unsigned index);
- 3 const char&
operator [] (unsigned index) const;
- 4 char&
operator [] (signed index);
- 5 const char&
operator [] (signed index) const;
- 6 const char&
operator [] (unsigned long index) const;
- 7 char&
operator [] (signed long index);
- 8 const char&
operator [] (signed long index) const;

| Exceptions | |
|----------------|--|
| InvalidRequest | 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. |

IString

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.

```
1 IString  
  subString( unsigned startPos ) const;
```

```
2 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 $0 \leq x \leq startPos$. 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.

```
1 unsigned
  lastIndexOf(
    const char* pString,
    unsigned startPos = ( unsigned ) UINT_MAX ) const;
```

```
2 unsigned
  lastIndexOf(
    const IString& aString,
    unsigned startPos = ( unsigned ) UINT_MAX ) const;
```

```
3 unsigned
  lastIndexOf(
    char aCharacter,
    unsigned startPos = ( unsigned ) UINT_MAX ) const;
```

```
4 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.

IString

- 1** unsigned
 lastIndexOfAnyBut(
 const IString& validChars,
 unsigned startPos = (unsigned) UINT_MAX) const;
- 2** unsigned
 lastIndexOfAnyBut(
 const char* pValidChars,
 unsigned startPos = (unsigned) UINT_MAX) const;
- 3** unsigned
 lastIndexOfAnyBut(
 char validChar,
 unsigned startPos = (unsigned) UINT_MAX) const;
- 4** 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.

- 1** unsigned
 lastIndexOfAnyOf(
 const char* pSearchChars,
 unsigned startPos = (unsigned) UINT_MAX) const;
- 2** unsigned
 lastIndexOfAnyOf(
 const IString& searchChars,
 unsigned startPos = (unsigned) UINT_MAX) const;

IString

```
3 unsigned
  lastIndexOfAnyOf(
    char searchChar,
    unsigned startPos = ( unsigned ) UINT_MAX ) const;

4 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;
```

IString

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 `isctrl()` 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}`.

IString

```
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 white-space characters. White-space 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, white-space 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.

IString

```
long  
asInt() const;
```

asLongLong Returns the number that the string represents as a long 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 long long, this function returns the maximum long long, not the larger value.

This function only exist on platforms with long long support.

```
long long  
asLongLong() const;
```

asUnsigned Returns, as an unsigned long, the integer that the string represents.

```
unsigned long  
asUnsigned() const;
```

asUnsignedLongLong

Returns, as an unsigned long long, the long long that the string represents.

Note: This member function only exist on platforms with long long support.

```
unsigned long long  
asUnsignedLongLong() 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.

IString

```
operator unsigned char *() const;
```

Word Operations

These members operate on a string as a collection of words separated by white-space 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. 546) and `IString::removeWords` obtain the same result but do not affect the `String` to which they are applied.

```
1 IString&  
   removeWords( unsigned firstWord );
```

IString

```
2 IString&
  removeWords( unsigned firstWord,
               unsigned numWords );
```

```
3 static IString
  removeWords( const IString& aString,
               unsigned startWord );
```

```
4 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. 545) obtain the same result but do not affect the String to which they are applied.

```
1 static IString
  space( const IString& aString,
         unsigned numSpaces = 1,
         char spaceChar = ' ' );
```

```
2 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;
```


IString

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.

```
1 IString
  words( unsigned firstWord,
         unsigned numWords ) const;
```

```
2 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.
```

IString

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 a 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 );
```

IString

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 the 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. 508) object
- An IStringTest (p. 573) object

The default is white space.

```
1 IString&
    strip( const char* pChar,
           unsigned len,
           IStringEnum::StripMode mode );
```

```
2 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.

IString

```
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. 525).

```
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.

- 1** IString&
initBuffer(double aDouble);
- 2** IString&
initBuffer(const void* p1,
 unsigned len1,
 const void* p2 = 0,
 unsigned len2 = 0,
 const void* p3 = 0,
 unsigned len3 = 0,
 char padChar = 0);
- 3** IString&
initBuffer(long aLong);

IString

4 IString&
initBuffer(unsigned long anUnsignedLong);

5 IString&
initBuffer(long long aLongLong);

6 IString&
initBuffer(unsigned long long anUnsignedLongLong);

setBuffer Sets the private data member to point to a new IBuffer (p. 340) object.

IString&
setBuffer(IBuffer* ibuff);

Queries

Use these members to access general information about the string.

buffer Returns the address of the IBuffer (p. 340) 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* pChar);

nullBuf Return the value of nullBuffer.

static const char*
nullBuf();

IString

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 global (or wildcard) characters, true is returned.

- You can use the first global character to specify that 0 or more arbitrary characters are accepted. The default global 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 global character to specify that a single arbitrary character is accepted. The default global 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;
```



This value is "2147483647" on OS/2 with 32-bit unsigned long integers.

null A string that contains no element.

```
static const char
*null;
```

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:

- bitAnd
- bitOr
- bitExclusiveOr
- and
- or
- exclusiveOr

and, or, exclusiveOr has the same value as bitAnd, bitOr, bitExclusiveOr and are slated for removal in the next release.

IString

IndexType `typedef enum { charIndex , wordIndex } IndexType;`

Use these enumerators to 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

Associated Globals

operator != `Boolean operator !=(const IString& string1, const char* pString2);`

If the strings are not bitwise identical, true is returned. This operator handles the following form of the comparison:

string1!=pString2

The first operand is an IString and the second is a NULL-terminated character string.

operator != `Boolean operator !=(const char* pString1, const IString& string2);`

If the strings are not bitwise identical, true is returned. This operator handles the following form of the comparison:

pString1!=string2

The first operand is a null-terminated character string and the second is an IString.

operator != `Boolean operator !=(const IString& string1, const IString& string2);`

If the strings are not bitwise identical, true is returned. This operator handles the following form of the comparison:

string1!=string2

Both operands are of type IString.

operator & `IString operator &(const char* pString, const IString& aString);`

IString

Performs bitwise AND. This operator handles the following form of the comparison:

pString & aString

The first operand is a NULL-terminated character string and the second is an IString.

operator + IString operator +(const char* pString, const IString& aString);

Concatenates a null-terminated character string and the second is an IString.

operator < Boolean operator <(const IString& string1, const char* pString2);

If the first string is less than the second, using the standard collating scheme (memcmp), true is returned. This operator handles the following form of the comparison:

string1 < pString2

The first operand is an IString and the second is a NULL-terminated character string.

Note: This operator is not locale-sensitive. Because it uses memcmp 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.

operator < Boolean operator <(const char* pString1, const IString& string2);

If the first string is less than the second, using the standard collating scheme (memcmp), true is returned. This operator can handles the following form of the comparison:

pString1 < string2

The first operand is a null-terminated character string and the second is an IString.

Note: This operator is not locale-sensitive. Because it uses memcmp 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.

operator < Boolean operator <(const IString& string1, const IString& string2);

IString

If the first string is less than the second, using the standard collating scheme (memcmp), true is returned. This operator handles the following form of the comparison:

string1 < string2

Both operands are of type IString.

Note: This operator is not locale-sensitive. Because it uses memcmp 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.

operator << ostream& operator <<(ostream& aStream, const IString& aString);

Puts an IString's contents to an output stream.

operator <= Boolean operator <=(const IString& string1, const char* pString2);

Equivalent to (string1 < string2) || (string1 == string2). This operator handles the following form of the comparison:

string1 <= pString2

The first operand is an IString and the second is a null-terminated character string.

Note: This operator is not locale-sensitive. Because it uses memcmp 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.

operator <= Boolean operator <=(const IString& string1, const IString& string2);

Equivalent to (string1 < string2) || (string1 == string2). This operator handles the following form of the comparison:

string1 <= string2

Both operands are of type IString.

Note: This operator is not locale-sensitive. Because it uses memcmp 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.

IString

operator <= Boolean operator <=(const char* pString1, const IString& string2);

Equivalent to (string1 < string2) || (string1 == string2). This operator can handle the following form of the comparison:

pString1 <= string2

The first operand is a null-terminated character string and the second is an IString.

Note: This operator is not locale-sensitive. Because it uses memcmp 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.

operator == Boolean operator ==(const char* pString1, const IString& string2);

If the strings are bitwise identical, true is returned. This operator handles the following form of the comparison:

pString1 == string2

The first operand is a null-terminated character string and the second is an IString.

operator == Boolean operator ==(const IString& string1, const char* pString2);

If the strings are bitwise identical, true is returned. This operator handles the following form of the comparison:

string1 == pString2

The first operand is an IString and the second is a null-terminated character string.

operator == Boolean operator ==(const IString& string1, const IString& string2);

If the strings are bitwise identical, true is returned. This operator handles the following form of the comparison:

string1 == string2

Both operands are of type IString.

operator > Boolean operator >(const IString& string1, const char* pString2);

Equivalent to !(string1 <= string2). This operator handles the following form of the comparison:

IString

string1 > pString2

The first operand is an IString and the second is a null-terminated character string.

Note: This operator is not locale-sensitive. Because it uses memcmp 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.

operator > Boolean operator `>(const IString& string1, const IString& string2);`

Equivalent to `!(string1 <= string2)`. This operator handles the following form of the comparison:

string1 > string2

Both operands are of type IString.

Note: This operator is not locale-sensitive. Because it uses memcmp 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.

operator > Boolean operator `>(const char* pString1, const IString& string2);`

Equivalent to `!(string1 <= string2)`. This operator handles the following form of the comparison:

pString1 > string2

The first operand is a NULL-terminated character string and the second is an IString.

Note: This operator is not locale-sensitive. Because it uses memcmp 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.

operator >= Boolean operator `>=(const char* pString1, const IString& string2);`

Equivalent to `!(string1 < string2)`. This operator handles the following form of the comparison:

IString

pString1 >= string2

The first operand is a null-terminated character string and the second is an IString.

Note: This operator is not locale-sensitive. Because it uses memcmp 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.

operator >= Boolean operator >=(const IString& string1, const IString& string2);

Equivalent to !(string1 < string2). This operator handles the following form of the comparison:

string1 >= string2

Both operands are of type IString.

Note: This operator is not locale-sensitive. Because it uses memcmp 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.

operator >= Boolean operator >=(const IString& string1, const char* pString2);

Equivalent to !(string1 < string2). This operator handles the following form of the comparison:

string1 >= pString2

The first operand is an IString and the second is a null-terminated character string.

Note: This operator is not locale-sensitive. Because it uses memcmp 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.

operator >> istream& operator >>(istream& aStream, IString& aString);

Puts the next white-space-delimited word from an input stream into an IString.

operator ^ IString operator ^(const char* pString, const IString& aString);

IString

Performs bitwise XOR. This operator handles the following form:

pString ^ aString

The first operand is a NULL-terminated character string and the second is an IString.

operator | IString operator |(const char* pString, const IString& aString);

Performs bitwise OR. This operator handles the following form:

pString | aString

The first operand is a null-terminated character string and the second is an IString.



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 User Interface Class Library places these enumeration types here so they can easily be shared between code that implements the classes IString (p. 508), IBuffer (p. 340), and IDBCSBuffer (p. 376).

Nested Type Definitions

CharType `typedef enum { sbcs , dbcs1 = 1 , mbcs1 = 1 , dbcs2 = 2 ,
mbcs2 = 2 , mbcs3 = 3 , mbcs4 = 4 } CharType;`

Use these enumerators to 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;`

Use this enumeration to define 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 | 568 | operator >> | 564 |
| | | operator << | 563 | ~IStringParser | 564 |

The IStringParser class parses the content of an IString (p. 508) and places 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.

IStringParser

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 );
```

IStringParser

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 User Interface Class 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. 571) 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. 568) to specify the parsing token.

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 the following:

- A string that defines the text to be parsed
- An existing parser object (copy constructor)

You 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. Because they hold references to operands, it is unwise to permit the objects to persist beyond the scope of those operands.

~IStringParser

~IStringParser();

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.

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 User Interface Class 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 >>(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.

3 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.

4 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.

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;
```

IStringParser

results 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 User Interface Class 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 >>(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
```

IStringParser

```
// token1 == "token1".
```

operator >> Parses the text string. The right-shift operator is the primary function for parsing the text string. The User Interface Class 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.

```
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.
```

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 the following:

- A string that defines the text to be parsed
- An existing parser object (copy constructor)

You 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. Because they hold references to operands, it is unwise to permit the objects to persist beyond the scope of those operands.

IStringParser

IStringParser

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 571)

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.

Associated Globals

operator << `IStringParser operator <<(const IString& text, unsigned long position);`

Implicitly constructs an IStringParser object from a string and performs absolute positioning on the object.

IStringParser

operator >> IStringParser operator >>(const IString& text, IStringParser::Command command);

Implicitly constructs an IStringParser object from a string and applies the IStringParser::Command to the object.

operator >> IStringParser operator >>(const IString& text, int position);

Implicitly constructs an IStringParser object from a string and performs relative positioning on the object.

operator >> IStringParser operator >>(const IString& text, const IStringTest& test);

Implicitly constructs an IStringParser object from a string and performs pattern matching based on the IStringTest object.

operator >> IStringParser operator >>(const IString& text, const char* pattern);

Implicitly constructs an IStringParser object from a string and performs pattern matching on the object.

operator >> IStringParser operator >>(const IString& text, IString& token);

Implicitly constructs an IStringParser object from a string and parses the text into the specified token string object.

operator >> IStringParser operator >>(const IString& text, const IString& pattern);

Implicitly constructs an IStringParser object from a string and performs pattern matching on the object.

operator >> IStringParser operator >>(const IString& text, char pattern);

Implicitly constructs an IStringParser object from a string and performs pattern matching on the object.

operator >> IStringParser operator >>(const IString& text, unsigned long position);

Implicitly constructs an IStringParser object from a string and performs relative positioning on the object.

IStringParser

operator >> IStringParser operator >>(const IString& text,
 const IStringParser::SkipWords& skipObject);

Implicitly constructs an IStringParser object from a string and skips the number of words specified by the IStringParser::SkipWords object.



IStringParser::SkipWords

Derivation IBase
 IStringParser::SkipWords

Inherited By None.

Header File
 istparse.hpp

| Members | Member | Page |
|----------------|---------------|-------------|
| | Constructor | 571 |
| | numberOfWords | 571 |
| | SkipWords | 571 |

The nested class IStringParser::SkipWords skips 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. 562).

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. 508) and place portions of the string into other strings.

SkipWords 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.

IStringParser::SkipWords

```
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 | 574 | type | 575 |
| | data | 575 | ~IStringTest | 574 |
| | test | 574 | | |

The IStringTest class defines the basic protocol for test objects that you can pass to IString (p. 508) functions or IOString (p. 287) functions 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 User Interface Class Library provides a derived template class, IStringTestMemberFn (p. 577), 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. 574) 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.



If I18N sementic is turned on, the integer parameter passed to the test function will be the wide-character representation of the character being tested.

Public Functions

IStringTest

Constructors

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. 574). 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( CFunction& cFunc );
```

Accepts a pointer to a C function.

~IStringTest

```
~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 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; void *user; }
data;
```

type Data member FnType. FnType is an enumeration describing the various flavors of
 functions supported: user-defined, C, C++ static or nonmember function, C++
 member function, or 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.

IStringTest

CFunction `typedef ICStrTestFn CFunction;`

Pointer to the C function that accepts an integer parameter and returns Boolean.



IStringTestMemberFn

Derivation IBase
 IVBase
 IStringTest
 IStringTestMemberFn

Inherited By None.

Header File
 istrtest.hpp

| Members | Member | Page |
|----------------|---------------|-------------|
| | Constructor | 578 |
| | test | 578 |

The IStringTestMemberFn class is a template class that provides an IStringTest-type wrapper for particular C++ member functions. You can use such member functions in conjunction with functions from IString (p. 508) and IString (p. 287) that accept an IStringTest (p. 573) 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 wrapped.

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 non-const member functions. You must specify a non-const member function as the first parameter.

Both constructors for the object require the following:

- An object of the class T (non-const object for non-const member functions).
- A pointer to a member function of the class T. The User Interface Class Library applies this member function to the specified object to test each character passed to the test member of

IStringTestMemberFn

this class. The member function must accept a single integer parameter and return a Boolean.

IStringTestMemberFn

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

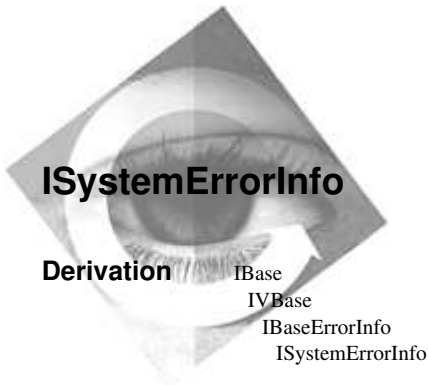
| IStringTest | | |
|-------------|------|--|
| data | type | |

| 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.



Inherited By None.

Header File

iexcept.hpp

Members

| Member | Page | Member | Page |
|-----------------------|------|-------------------|------|
| Constructor | 581 | text | 582 |
| errorId | 581 | throwSystemError | 582 |
| isAvailable | 581 | ~ISystemErrorInfo | 581 |
| operator const char * | 582 | | |

The ISystemErrorInfo class represents error information that you can include in an exception object. When an operating 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. 394).

The User Interface Class 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 IBaseErrorInfo::ExceptionType (p. 319), which specifies the type of exception this macro creates
- One of the values of the enumeration IException::Severity (p. 403), which specifies the severity of the exception

This macro generates code that calls throwSystemError (p. 582), 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

ISystemErrorInfo



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

Constructors

You can construct and destruct objects of this class. You cannot copy or assign objects of this class.

ISystemErrorInfo

```
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 User Interface Class Library provides the following default text: "No error text is available."

systemErrorId

The error ID identifying an operating system error.

systemFunctionName

The name of the failing system call that returned the error ID. If you specify *systemFunctionName*, the constructor prefixes it to the error text. Optional.

~ISystemErrorInfo

```
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.

ISystemErrorInfo

```
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

Used by the ITHROWSYSTEMERROR macro, this 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. 406) object containing the following:

- Function name
- File name
- Line number where the function is called

name Use the enumeration IBaseErrorInfo::ExceptionType (p. 319) to specify the type of the exception. The default is accessError.

severity Use the enumeration IException::Severity (p. 403) to specify the severity of the error. The default is recoverable.

ISystemErrorInfo

```
static void  
    throwSystemError(  
        unsigned long systemErrorId,  
        const char* functionName,  
        const IExceptionLocation& location,  
        IErrorInfo::ExceptionType name = accessError,  
        IException::Severity severity = recoverable );
```

Inherited Public Functions

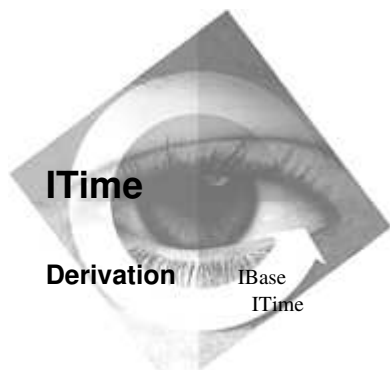
| IBaseErrorInfo | | |
|----------------|-------------|-----------------------|
| errorId | isAvailable | operator const char * |

| IVBase | | |
|-------------|----------|--|
| asDebugInfo | asString | |

| IBase | | |
|-------------|-------------|----------------|
| asDebugInfo | messageFile | setMessageFile |
| asString | messageText | version |

Inherited Protected Data

| IBase | | |
|-------------|---------------|--|
| recoverable | unrecoverable | |



Inherited By None.

Header File
itime.hpp

| Members | | Member | Page | Member | Page |
|---------|--|-------------|------|-------------|------|
| | | Constructor | 585 | operator += | 587 |
| | | asICnrTime | 588 | operator - | 587 |
| | | asSeconds | 588 | operator -= | 587 |
| | | asString | 586 | operator < | 585 |
| | | hours | 588 | operator <= | 585 |
| | | initialize | 589 | operator == | 585 |
| | | minutes | 588 | operator > | 585 |
| | | now | 586 | operator >= | 585 |
| | | operator != | 584 | seconds | 588 |
| | | operator + | 587 | | |

The ITime class represents units of time (hours, minutes, and seconds) as portions of days and provides 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. 365).

The ITime class returns locale-sensitive information, based on the current locale defined at run time. See the description of the standard C function setlocale in your specific platform's 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.

ITime

```
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.

ITime

```
1 ITime( const ITime& aTime );
```

ITime

Use this constructor to copy another ITime object.

2 ITime();

Use this constructor to return the current time; it is 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 ICnrTime& cnrTime);

Use this constructor to construct an ITime object from a container details ICnrTime 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 for more information about the strftime function.

ITime

The conversion specifiers that apply to ITime and their meanings are listed in the following table. IDate::asString (p. 369) 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 %. |

```
IStrString  
  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

```
ITime&
operator --( const ITime& aTime );
```

Time Queries

Use these members to access the seconds, minutes, and hours of an ITime object.

asICnrTime Returns the time as a container ICnrTime structure.

```
ICnrTime
asICnrTime() const;
```

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

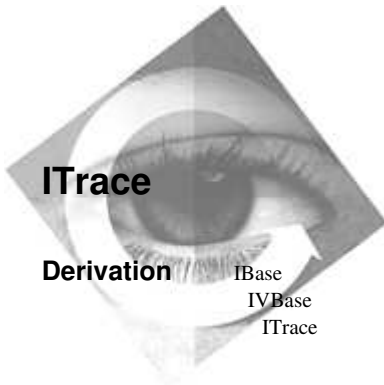
Use these members to initialize objects of this class.

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 | 592 | threadId | 597 |
| disableTrace | 593 | traceDestination | 594 |
| disableWriteLineNumber | 593 | write | 594 |
| disableWritePrefix | 593 | writeFormattedString | 596 |
| enableTrace | 593 | writeString | 596 |
| enableWriteLineNumber | 594 | writeToFile | 595 |
| enableWritePrefix | 594 | writeToQueue | 595 |
| isTraceEnabled | 593 | writeToStandardError | 595 |
| isWriteLineNumberEnabled | 594 | writeToStandardOutput | 595 |
| isWritePrefixEnabled | 594 | ~ITrace | 593 |

The ITrace class provides module tracing within the User Interface Class 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. 406)

The Application Support 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:

ITrace

- 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 User Interface Class 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()*

ITrace

- 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.



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`.



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

Constructors

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.

ITrace

```
ITrace( const char* traceName = 0,  
        long lineNumber = 0 );
```

You pass the optional parameters to gain the following trace behavior:

traceName If you specify *traceName*, the name is written on construction and again on destruction. Optional.

Warning: If you pass an `IString` (p. 508) object 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 `IString` objects.

ITrace

lineNumber

The line number where the trace statement occurred. Optional.

~ITrace

```
~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();
```

ITrace

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. 597).

```
static ITrace::Destination  
    traceDestination();
```

write

Writes the specified text.

1 static void
 write(const IString& text);
text The text to write as an IString (p. 508).

2 static void
 write(const char* text);
text The text to write as a character string.

writeToFile Set the location for output to a file. Using this function is equivalent to setting the environment variable *ICLUI_TRACETO=FILE*.

In order for this to work another environment variable ICLUI_TRACEFILE needs to be set.

```
static void
writeToFile();
```

writeToQueue

Sets the location for output to \\QUEUES\PRINTF32 on OS/2 environment.

Sets the location for output to \\MAILSLOT\PRINTF32 on Win32 environment.

Sets the location for output to Job log on OS/400 environment.

```
static void
writeToQueue();
```



In AIX, this member function is equivalent to writeToStandardOutput (p. 595).

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();
```

ITrace

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 User Interface Class 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();
```



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,
 file
};

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).
- file** Sends the trace data to a file specified by environment variable ICLUI_TRACEFILE.

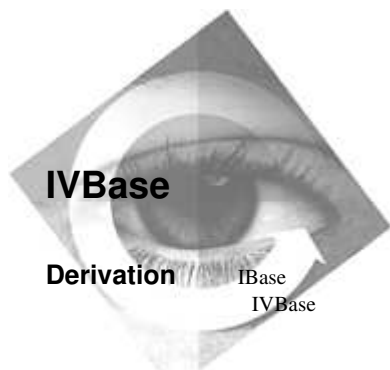
When used on the following platforms, the queue enumerator is not supported, and queue tracing goes to stdout:

- AIX
- Solaris
- MVS

When used on AS/400, sending the trace data to queue means sending to the job log.



AIX does not support the queue enumerator. If the trace destination is queue, tracing goes to stdout.



IVBase

Derivation

IBase
IVBase

Inherited By

| | |
|---------------------------------|----------------------------|
| IAcceleratorTable::Cursor | IHandler |
| IApplication | IMenu::Cursor |
| IBaseComboBox::Cursor | IMessageBox |
| IBaseErrorInfo | IMMAudioCDContents |
| IBaseListBox::Cursor | IMMAudioCDContents::Cursor |
| IBaseStream | IMMSpeed |
| IBidiSettings | IMMTime |
| IBuffer | IModel |
| IClipboard | INotebook::Cursor |
| IClipboard::Cursor | INotebook::PageSettings |
| IColor | INotifier |
| IComponent | IObserver |
| IComponentStationery | IObserverList |
| IContainerColumn | IObserverList::Cursor |
| IContainerControl::ColumnCursor | IProfile |
| IContainerControl::CompareFn | IProfile::Cursor |
| IContainerControl::FilterFn | IRefCounted |
| IContainerControl::Iterator | IResource |
| IContainerControl::ObjectCursor | IResourceLibrary |
| IContainerControl::TextCursor | IResourceLock |
| IContainerObject | IStringTest |
| IDMImage | ISubmenu::Cursor |
| IDMItemProvider | ITextSpinButton::Cursor |
| IDMRenderer | IThread |
| IDocumentStorage | IThread::Cursor |
| IEvent | ITimer |
| IFont | ITimer::Cursor |
| IFont::FaceNameCursor | IToolBar::FrameCursor |
| IFont::PointSizeCursor | IToolBar::WindowCursor |
| IGList::Cursor | ITrace |
| IGrabHandles | IWindow::ChildCursor |
| IGraphic | IWindow::ExceptionFn |
| IGraphicContext | |

Header File

ivbase.hpp

Members

| Member | Page |
|-------------|------|
| asDebugInfo | 599 |
| asString | 599 |
| ~IVBase | 599 |

IVBase

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. 308) 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 ostream, such as cout, cerr, or both. See asString (p. 599) and asDebugInfo (p. 599) for information about those functions.

Public Functions

Constructors

You can use the virtual destructor that this class provides to ensure that all derived classes' destructors are also virtual.

~IVBase

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 |

IVBase

| IVBase | | |
|----------|-------------|---------|
| asString | messageText | version |

Inherited Protected Data

| IVBase | | |
|-------------|---------------|--|
| recoverable | unrecoverable | |

Associated Globals

operator << ostream& operator <<(ostream& aStream, const IVBase& anObject);

Puts an object representation to an output stream. By default, this operator puts the asString representation.



IXLibErrorInfo

Derivation IBase
 IVBase
 IBaseErrorInfo
 IXLibErrorInfo

Inherited By None.

Header File
 iexcept.hpp

| Members | Member | Page | Member | Page |
|---------|-----------------------|------|-----------------|------|
| | Constructor | 602 | text | 603 |
| | errorId | 603 | throwXLibError | 603 |
| | isAvailable | 603 | ~IXLibErrorInfo | 602 |
| | operator const char * | 603 | | |

The IXLibErrorInfo class represents 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. 394).

The User Interface Class 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. 319) to specify the type of the exception. The default is accessError.

IXLibErrorInfo

severity Use the enumeration `IException::Severity` (p. 403) to specify the severity of the error. The default is recoverable.

This macro generates code that calls `throwXLibError` (p. 603), 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



The OS/2 release of the User Interface Class 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 releases of the library, this class is not supported.

Public Functions

Constructors

You can construct and destruct objects of this class. You cannot copy or assign objects of this class.

IXLibErrorInfo

| | | | |
|--|------------|-----------|--------------|
| <code>IXLibErrorInfo(const char* systemFunctionName = 0);</code> | <u>Win</u> | <u>PM</u> | <u>Motif</u> |
| | <i>N</i> | <i>N</i> | <i>Y</i> |

You can only construct objects of this class using the default constructor.

Note: If the constructor cannot load the error text, the User Interface Class Library provides the following default text: "No error text is available."

systemFunctionName

The name of the failing X Library function. If you specify *systemFunctionName*, the constructor prefixes it to the error text.
Optional.

~IXLibErrorInfo

| | | | |
|---|------------|-----------|--------------|
| <code>virtual ~IXLibErrorInfo();</code> | <u>Win</u> | <u>PM</u> | <u>Motif</u> |
| | <i>N</i> | <i>N</i> | <i>Y</i> |

IXLibErrorInfo

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.

| | | | |
|---|-------------------|------------------|---------------------|
| <code>virtual unsigned long errorId() const;</code> | <u>Win</u> | <u>PM</u> | <u>Motif</u> |
| | <i>N</i> | <i>N</i> | <i>Y</i> |

isAvailable If the error text is available, true is returned.

| | | | |
|---|-------------------|------------------|---------------------|
| <code>virtual Boolean isAvailable() const;</code> | <u>Win</u> | <u>PM</u> | <u>Motif</u> |
| | <i>N</i> | <i>N</i> | <i>Y</i> |

operator const char *

Returns the error text.

| | | | |
|---|-------------------|------------------|---------------------|
| <code>virtual operator const char *() const;</code> | <u>Win</u> | <u>PM</u> | <u>Motif</u> |
| | <i>N</i> | <i>N</i> | <i>Y</i> |

text Returns the error text.

| | | | |
|--|-------------------|------------------|---------------------|
| <code>virtual const char* text() const;</code> | <u>Win</u> | <u>PM</u> | <u>Motif</u> |
| | <i>N</i> | <i>N</i> | <i>Y</i> |

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. 406) object containing the following:

- Function name
- File name

IXLibErrorInfo

- Line number where the function is called

name Use the enumeration IBaseErrorInfo::ExceptionType (p. 319) to specify the type of the exception. The default is accessError.

severity Use the enumeration IException::Severity (p. 403) to specify the severity of the error. The default is recoverable.

```
static void                                     Win PM Motif  
throwXLibError(  
    const char* functionName,  
    const IExceptionLocation& location,  
    IErrorInfo::ExceptionType name = accessError,  
    IException::Severity severity = recoverable );
```

Win PM Motif
N *N* *Y*

Inherited Public Functions

| IBaseErrorInfo | | |
|----------------|-------------|-----------------------|
| errorId | isAvailable | operator const char * |

| IVBase | | |
|-------------|----------|--|
| asDebugInfo | asString | |

| IBase | | |
|-------------|-------------|----------------|
| asDebugInfo | messageFile | setMessageFile |
| asString | messageText | version |

Inherited Protected Data

| IBase | | |
|-------------|---------------|--|
| recoverable | unrecoverable | |

Part 8. Data Access Builder C++ Classes and Exception Classes

Use the database access classes to connect to and to disconnect from your DB2 and other supported databases, and to apply transactions against those databases.

This section describes the C++ version of the database access classes. Use these classes when you generate native C++ code or Visual Builder parts.

| | |
|---|-----|
| IDAException | 607 |
| Public Functions | 608 |
| Inherited Public Functions | 609 |
| Inherited Public Data | 609 |
| IDatastore | 610 |
| Public Functions | 611 |
| Inherited Public Functions | 614 |
| Inherited Protected Functions | 614 |
| Public Data | 615 |
| Inherited Public Data | 615 |
| Inherited Protected Data | 615 |
| IDatastoreBase | 616 |
| Public Functions | 617 |
| Inherited Public Functions | 621 |
| Protected Functions | 621 |
| Inherited Protected Functions | 622 |
| Public Data | 622 |
| Inherited Public Data | 623 |
| Inherited Protected Data | 623 |
| IDatastoreDB2 | 624 |
| Public Functions | 624 |
| Inherited Public Functions | 629 |
| Inherited Protected Functions | 630 |
| Public Data | 630 |
| Inherited Public Data | 631 |
| Inherited Protected Data | 631 |
| IDatastoreODBC | 632 |
| Public Functions | 632 |

| | |
|---|-----|
| Inherited Public Functions | 637 |
| Inherited Protected Functions | 638 |
| Public Data | 638 |
| Inherited Public Data | 639 |
| Inherited Protected Data | 639 |
| IPersistentObject | 640 |
| Public Functions | 640 |
| Inherited Public Functions | 643 |
| Protected Functions | 644 |
| Inherited Protected Data | 644 |
| IPOManager | 645 |
| Public Functions | 645 |
| Protected Functions | 646 |



IDAException

Derivation IException
 IDAException

Inherited By None.

Header File
 idsexc.hpp

| Members | Member | Page | Member | Page |
|---------|---------------|------|------------|------|
| | errorAsString | 608 | errorState | 608 |
| | errorCode | 608 | getSqlca | 609 |
| | errorProvided | 608 | | |

The following are the C++ exception classes in Data Access Builder:

IDAException

The base class for all exception classes in Data Access Builder.

IDAAccessError

An object of this class is thrown when an error occurs during an attempt to access data in the datastore.

IDAAdaptorException

The base class for a number of exception classes that create and throw objects when database access errors occur.

IDAConnectionInUse

An object of this class is thrown when a connection is attempted using a connection that is already in use.

IDAConnectionNotOpen

An object of this class is thrown when an operation that requires a connection is attempted before a connection has been established. An example is a call to disconnect before a connection was made.

IDAConnectFailed

An object of this class is thrown when a database connection attempt fails.

IDADataObjectAlreadyExists

An object of this class is thrown when the data object specified in an add operation already exists in the datastore.

IDADataObjectAttributeError

An object of this class is thrown when an attempt is made to set an attribute with an invalid value. An example is setting a string attribute longer than the database permits.

IDAException

IDADataObjectInvalid

An object of this class is thrown when an attempt to update or to delete an object results in changes to more than one row in the datastore.

IDADataObjectNotFound

An object of this class is thrown when the data object specified in a retrieve, update, or delete operation cannot be found in the datastore.

IDADisconnectError

An object of this class is thrown when a disconnection error occurs.

IDALogoffFailed

An object of this class is thrown when a logoff fails.

IDALogonFailed

An object of this class is thrown when a logon attempt fails.

Public Functions

Queries

These members are used to access information about an object.

errorAsString

Returns the sqlcode and the sqlstate as a string. If errorProvided has a value of true, returns "SQLCODE: " + IString (sqlcode) + " SQLSTATE: " + IString (sqlstate,5); otherwise returns IString ().

| | | | |
|------------------------|-------------------|------------------|---------------------|
| IString | <u>Win</u> | <u>PM</u> | <u>Motif</u> |
| errorAsString() const; | <i>Y</i> | <i>N</i> | <i>N</i> |

errorCode

Returns the sqlcode at the time of exception. Valid only when errorProvided has a value of true.

| | | | |
|--------------------|-------------------|------------------|---------------------|
| long | <u>Win</u> | <u>PM</u> | <u>Motif</u> |
| errorCode() const; | <i>Y</i> | <i>N</i> | <i>N</i> |

errorProvided

A Boolean flag indicating whether the exception is the result of an error reported by the datastore.

| | | | |
|------------------------|-------------------|------------------|---------------------|
| Boolean | <u>Win</u> | <u>PM</u> | <u>Motif</u> |
| errorProvided() const; | <i>Y</i> | <i>N</i> | <i>N</i> |

errorState

Returns the sqlstate at the time of exception. Valid only when errorProvided has a value of true.

IDAException

```
const char*
    errorState() const;
```

| <u>Win</u> | <u>PM</u> | <u>Motif</u> |
|------------|-----------|--------------|
| <i>Y</i> | <i>N</i> | <i>N</i> |

getSqlca

Returns the sqlcode and the sqlstate information contained in the sqlca at the time of exception. This method is maintained for compatibility with Data Access Builder version 1.0.

```
struct DA_sqlca
    getSqlca() const;
```

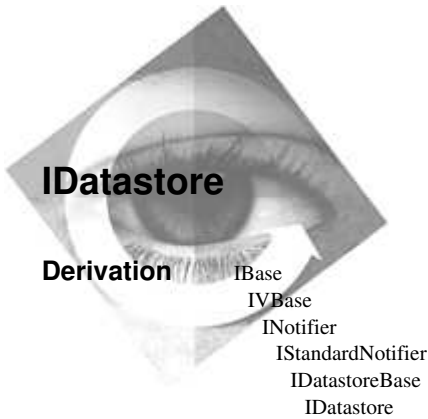
| <u>Win</u> | <u>PM</u> | <u>Motif</u> |
|------------|-----------|--------------|
| <i>Y</i> | <i>N</i> | <i>N</i> |

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 |



Inherited By None.

Header File
idsmcon.hpp

| Members | | | | |
|--------------------------|------|----------------------|------|--|
| Member | Page | Member | Page | |
| Constructor | 611 | readyId | 615 | |
| commit | 613 | rollback | 614 | |
| connect | 612 | setAuthentication | 613 | |
| disconnect | 612 | setDatastoreName | 613 | |
| enableShareModeExclusive | 613 | setUserName | 613 | |
| executeSQL | 611 | shareModeExclusiveId | 615 | |
| isShareModeExclusive | 612 | ~IDatastore | 611 | |

Use IDatastore with persistent objects that were generated using the embedded SQL option to access DB2 version 1.2 or 2.1 embedded SQL. Data Access Builder permits only one IDatastore instance connection to DB2 per application process. To access DB2 version 2.1 or later using CLI, use IDatastoreDB2 (p. 624).

Only one connection should be active at a time when using embedded SQL.

An IDatastore can be used with an IDatastoreODBC, and an IDatastoreDB2 can be used with an IDatastoreODBC, but an IDatastore *cannot* be used with an IDatastoreDB2.

For DB2 version 1.2 and 2.1 embedded SQL, if *userName* and *authentication* are both specified, these attributes are used to request access to the database; otherwise, an attempt is made to access the database without this information.

Important

When IDatastore for embedded SQL is used, the bind file associated with the class library (*.cppwac1.bnd) must be bound to each database to which an application will be connecting. This file is located in the bnd directory of the VisualAge for C++ product directory (X:\ibmcpp\bnd). However, if Data Access Builder is used to create any embedded SQL classes for the database, the bind step is done automatically.



Data Access Builder does not support DB2 version 1.2 on Windows platforms.

Public Functions

Constructors

IDatastore has two public constructors and a destructor.

IDatastore This class has two constructors; one takes no parameters, and the other takes one required parameter and two optional parameters.

| | | | | |
|----------|---|-----------------|----------------|-------------------|
| 1 | IDatastore(); | <u>Win</u> Y | <u>PM</u> Y | <u>Motif</u> N |
| 2 | IDatastore(const char* datastoreName, const char* userName = "", const char* authentication = ""); | <u>Win</u> Y | <u>PM</u> Y | <u>Motif</u> N |

~IDatastore

The IDatastore destructor frees space allocated by the constructor, and executes a disconnect if the application has not explicitly done so.

| | | | |
|---------------------------|-----------------|----------------|-------------------|
| virtual ~IDatastore(); | <u>Win</u> Y | <u>PM</u> Y | <u>Motif</u> N |
|---------------------------|-----------------|----------------|-------------------|

Data Manipulation

This member is used to execute a dynamic SQL statement.

executeSQL For a description of this function, see IDatastoreBase::executeSQL (p. 617).

Note: Methods implemented in the base class return a reference to the calling class type.

IDatastore

| | | | |
|---|-------------------|------------------|---------------------|
| <code>virtual IDatastore& executeSQL(const char* aCmd);</code> | <u>Win</u> | <u>PM</u> | <u>Motif</u> |
| | <i>Y</i> | <i>N</i> | <i>N</i> |

Database Connection

These members are used to connect to or to disconnect from a datastore.

connect For a description of this function, see `IDatastoreBase::connect` (p. 617).

Note: Methods implemented in the base class return a reference to the calling class type.

| | | | | |
|----------|--|-------------------|------------------|---------------------|
| 1 | <code>virtual IDatastore& connect(const char* userName, const char* authentication);</code> | <u>Win</u> | <u>PM</u> | <u>Motif</u> |
| | | <i>Y</i> | <i>N</i> | <i>N</i> |

| | | | | |
|----------|---|-------------------|------------------|---------------------|
| 2 | <code>virtual IDatastore& connect();</code> | <u>Win</u> | <u>PM</u> | <u>Motif</u> |
| | | <i>Y</i> | <i>Y</i> | <i>N</i> |

| | | | | |
|----------|---|-------------------|------------------|---------------------|
| 3 | <code>virtual IDatastore& connect(const char* datastoreName, const char* userName, const char* authentication);</code> | <u>Win</u> | <u>PM</u> | <u>Motif</u> |
| | | <i>Y</i> | <i>Y</i> | <i>N</i> |

| | | | | |
|----------|---|-------------------|------------------|---------------------|
| 4 | <code>virtual IDatastore& connect(const char* datastoreName);</code> | <u>Win</u> | <u>PM</u> | <u>Motif</u> |
| | | <i>Y</i> | <i>N</i> | <i>N</i> |

disconnect For a description of this function, see `IDatastoreBase::disconnect` (p. 618).

Note: Methods implemented in the base class return a reference to the calling class type.

| | | | |
|--|-------------------|------------------|---------------------|
| <code>virtual IDatastore& disconnect();</code> | <u>Win</u> | <u>PM</u> | <u>Motif</u> |
| | <i>Y</i> | <i>Y</i> | <i>N</i> |

Queries

These members are used to access information about an object.

isShareModeExclusive

Returns true if the exclusive mode has been enabled, or false if it has been disabled for this connection.

IDatastore

```
virtual Boolean  
    isShareModeExclusive();
```

| <u>Win</u> | <u>PM</u> | <u>Motif</u> |
|------------|-----------|--------------|
| <i>Y</i> | <i>N</i> | <i>N</i> |

State Modifiers

These members are used to modify the state of an object.

enableShareModeExclusive

Enables or disables the exclusive mode in a database connection. Enabling this option specifies exclusive access to the database; no other user will be able to connect to the database until the original connection is reset.

```
virtual IDatastore&  
    enableShareModeExclusive( Boolean enable = true);
```

| <u>Win</u> | <u>PM</u> | <u>Motif</u> |
|------------|-----------|--------------|
| <i>Y</i> | <i>N</i> | <i>N</i> |

setAuthentication

Sets the authentication.

```
virtual IDatastore&  
    setAuthentication( const char* aAuthentication);
```

| <u>Win</u> | <u>PM</u> | <u>Motif</u> |
|------------|-----------|--------------|
| <i>Y</i> | <i>Y</i> | <i>N</i> |

setDatastoreName

Sets the datastore name that is used when a connection is established.

```
virtual IDatastore&  
    setDatastoreName( const char* aDatastoreName);
```

| <u>Win</u> | <u>PM</u> | <u>Motif</u> |
|------------|-----------|--------------|
| <i>Y</i> | <i>Y</i> | <i>N</i> |

setUserName

Sets the user name.

```
virtual IDatastore&  
    setUserName( const char* aUserName);
```

| <u>Win</u> | <u>PM</u> | <u>Motif</u> |
|------------|-----------|--------------|
| <i>Y</i> | <i>Y</i> | <i>N</i> |

Transaction Handling

These members are used to commit or to roll back transactions.

commit

For a description of this function, see IDatastoreBase::commit (p. 620).

Note: Methods implemented in the base class return a reference to the calling class type.

IDatastore

```
virtual IDatastore&  
    commit();
```

| <u>Win</u> | <u>PM</u> | <u>Motif</u> |
|------------|-----------|--------------|
| <i>Y</i> | <i>Y</i> | <i>N</i> |

rollback

For a description of this function, see IDatastoreBase::rollback (p. 620).

Note: Methods implemented in the base class return a reference to the calling class type.

```
virtual IDatastore&  
    rollback();
```

| <u>Win</u> | <u>PM</u> | <u>Motif</u> |
|------------|-----------|--------------|
| <i>Y</i> | <i>Y</i> | <i>N</i> |

Inherited Public Functions

| IDatastoreBase | | |
|----------------|---------------|-------------------|
| asString | datastoreName | rollback |
| authentication | disconnect | setAuthentication |
| commit | executeSQL | setDatastoreName |
| connect | isConnected | setUserName |

| IStandardNotifier | | |
|---------------------|--------------------|--------------------------|
| disableNotification | enableNotification | isEnabledForNotification |

| INotifier | | |
|---------------------|--------------------|--------------------------|
| disableNotification | enableNotification | isEnabledForNotification |

| IVBase | | |
|-------------|----------|--|
| asDebugInfo | asString | |

| IBase | | |
|-------------|-------------|----------------|
| asDebugInfo | messageFile | setMessageFile |
| asString | messageText | version |

Inherited Protected Functions

| IStandardNotifier | | |
|-------------------|-----------------|--------------|
| addObserver | notifyObservers | observerList |

IDatastore

| INotifier | | |
|-------------|-----------------|--------------|
| addObserver | notifyObservers | observerList |

Public Data

Notification Event Descriptions

These INotificationId strings are used for all notifications that this class provides to its observers.

readyId Notification identifier sent when a part is initialized.

| | | | |
|--|-----------------|----------------|-------------------|
| static const INotificationId readyId; | <u>Win</u> Y | <u>PM</u> N | <u>Motif</u> N |
|--|-----------------|----------------|-------------------|

shareModeExclusiveld

Notification identifier sent when the *shareModeExclusive* attribute is modified.

| | | | |
|---|-----------------|----------------|-------------------|
| static const INotificationId shareModeExclusiveId; | <u>Win</u> Y | <u>PM</u> N | <u>Motif</u> N |
|---|-----------------|----------------|-------------------|

Inherited Public Data

| IDatastoreBase | | |
|------------------|-----------------|---------------|
| AuthenticationId | DatastoreNameId | isConnectedId |
| connectedId | disconnectedId | transactedId |

| IStandardNotifier | | |
|-------------------|--|--|
| deleteId | | |

Inherited Protected Data

| IBase | | |
|-------------|---------------|--|
| recoverable | unrecoverable | |



Inherited By IDatastore
 IDatastoreDB2
 IDatastoreODBC

Header File
 idsmcbs.hpp

| Members | | | | |
|------------------|------|-------------------|------|--|
| Member | Page | Member | Page | |
| Constructor | 621 | executeSQL | 617 | |
| asString | 619 | isConnected | 619 | |
| authentication | 619 | isConnectedId | 623 | |
| AuthenticationId | 622 | rollback | 620 | |
| commit | 620 | setAuthentication | 620 | |
| connect | 617 | setDatastoreName | 620 | |
| connectedId | 622 | setUserName | 620 | |
| datastoreName | 619 | transactedId | 623 | |
| DatastoreNameId | 622 | userName | 619 | |
| disconnect | 618 | UserNameId | 623 | |
| disconnectedId | 622 | ~IDatastoreBase | 617 | |

IDatastoreBase is an abstract base class for all Data Access Builder C++ classes that represent datastore connections. IDatastoreBase provides client connection to the database, disconnection from the database, and commit and rollback of transactions. Objects of this class are not instantiated, but the class may be used as a generic reference or pointer to objects instantiated from the concrete derived classes. The datastore classes are enabled with the notification framework so that they can be used with applications generated with the Visual Builder.

The following attributes are used by IDatastoreBase:

- authentication
 Authentication is the password for the user identified by *userName*. An unspecified value is indicated by " ".
- datastoreName

IDatastoreBase

Name of the datastore to which the user wishes to connect. In DB2, *datastoreName* represents the name of the database; in ODBC, it represents the name of the datasource.

- `userName`

Name of the user to be passed to the datasource. This name and the password are used to complete authorization of the user. If unspecified (or set to " "), the *userName* is not passed to the datasource, and a connection attempt may fail, or the datasource may attempt to determine the user's authorization independently.

Public Functions

Constructors

IDatastoreBase has three protected constructors and a destructor.

~IDatastoreBase

```
virtual  
~IDatastoreBase();
```

| <u>Win</u> | <u>PM</u> | <u>Motif</u> |
|------------|-----------|--------------|
| <i>Y</i> | <i>N</i> | <i>N</i> |

Data Manipulation

This member is used to execute a dynamic SQL statement.

executeSQL Executes a dynamic SQL statement. If a **select** statement is passed in:

- IDatastore always returns an exception
- IDatastoreDB2 and IDatastoreODBC ignore any result other than an access error message.

```
virtual IDatastoreBase&  
executeSQL( const char* aCmd);
```

| <u>Win</u> | <u>PM</u> | <u>Motif</u> |
|------------|-----------|--------------|
| <i>Y</i> | <i>N</i> | <i>N</i> |

| Exceptions | |
|----------------|---|
| IDAAccessError | An object of this class is thrown when an error occurs during an attempt to access data in the datastore. |

Database Connection

These members are used to connect to or to disconnect from a datastore.

connect Connects to a datastore. If a connection already exists, performs a disconnect, and then reconnects using the current settings.

IDatastoreBase

| | | | | |
|----------|---------------------------------------|------------|-----------|--------------|
| 1 | virtual IDatastoreBase& connect(); | Win | PM | Motif |
| | | <i>Y</i> | <i>N</i> | <i>N</i> |

Connects to a datastore using the currently specified datastore settings.

| Exceptions | |
|------------------|---|
| IDAConnectFailed | An object of this class is thrown when a database connection attempt fails. |
| IDAAccessError | An object of this class is thrown when an error occurs during an attempt to access data in the datastore. |
| IDALogoffFailed | An object of this class is thrown when a logoff fails. |
| IDALogonFailed | An object of this class is thrown when a logon attempt fails. |

| | | | | |
|---|---|-----------------|----------------|-------------------|
| 2 | virtual IDatastoreBase& connect(const char* userName, const char* authentication); | <u>Win</u> Y | <u>PM</u> Y | <u>Motif</u> N |
|---|---|-----------------|----------------|-------------------|

Connects to a datastore using the specified input parameters and the current datastoreName setting. The values of userName and authentication are not saved.

| | | | | | | | | |
|-------------------|---|---|-------------------|------------------|---------------------|----------|----------|----------|
| 3 | <pre>virtual IDatastoreBase& connect(const char* datastoreName);</pre> | <table border="0"> <tr> <td><u>Win</u></td> <td><u>PM</u></td> <td><u>Motif</u></td> </tr> <tr> <td><i>Y</i></td> <td><i>Y</i></td> <td><i>N</i></td> </tr> </table> | <u>Win</u> | <u>PM</u> | <u>Motif</u> | <i>Y</i> | <i>Y</i> | <i>N</i> |
| <u>Win</u> | <u>PM</u> | <u>Motif</u> | | | | | | |
| <i>Y</i> | <i>Y</i> | <i>N</i> | | | | | | |

Connects to a datastore using the specified datastoreName and the current user ID and authentication settings.

| | | | | | | | | |
|-------------------|---|---|-------------------|------------------|---------------------|----------|----------|----------|
| 4 | <pre>virtual IDatastoreBase& connect(const char* datastoreName, const char* userName, const char* authentication);</pre> | <table border="0"> <tr> <td><u>Win</u></td> <td><u>PM</u></td> <td><u>Motif</u></td> </tr> <tr> <td><i>Y</i></td> <td><i>N</i></td> <td><i>N</i></td> </tr> </table> | <u>Win</u> | <u>PM</u> | <u>Motif</u> | <i>Y</i> | <i>N</i> | <i>N</i> |
| <u>Win</u> | <u>PM</u> | <u>Motif</u> | | | | | | |
| <i>Y</i> | <i>N</i> | <i>N</i> | | | | | | |

Connects to a datastore using the specified input parameters. The values of userName and authentication are not saved.

disconnect Closes the connection to a database.

| | | | |
|--|------------------------|-----------------------|--------------------------|
| virtual IDatastoreBase& disconnect(); | <u>Win</u> Y | <u>PM</u> Y | <u>Motif</u> N |
|--|------------------------|-----------------------|--------------------------|

IDatastoreBase

| Exceptions | |
|----------------------|--|
| IDAConnectionNotOpen | An object of this class is thrown when an operation that requires a connection is attempted before a connection has been established. An example is a call to disconnect before a connection was made. |
| IDALogoffFailed | An object of this class is thrown when a logoff fails. |
| IDADisconnectError | An object of this class is thrown when a disconnection error occurs. |

Queries

These members are used to access information about an object.

asString Returns a string representing the connection. The string contains the name of the datastore set with the constructor or with the *setDatastoreName* method.

| | | | |
|--------------------------------------|-----------------|----------------|-------------------|
| virtual IString asString() const; | <u>Win</u> Y | <u>PM</u> N | <u>Motif</u> N |
|--------------------------------------|-----------------|----------------|-------------------|

authentication

Returns the current authentication setting.

| | | | |
|--|-----------------|----------------|-------------------|
| virtual IString authentication() const; | <u>Win</u> Y | <u>PM</u> Y | <u>Motif</u> N |
|--|-----------------|----------------|-------------------|

datastoreName

Returns the current datastore name setting.

| | | | |
|---|-----------------|----------------|-------------------|
| virtual IString datastoreName() const; | <u>Win</u> Y | <u>PM</u> Y | <u>Motif</u> N |
|---|-----------------|----------------|-------------------|

isConnected Returns true if a connection to the database exists.

| | | | |
|-----------------------------------|-----------------|----------------|-------------------|
| virtual Boolean isConnected(); | <u>Win</u> Y | <u>PM</u> Y | <u>Motif</u> N |
|-----------------------------------|-----------------|----------------|-------------------|

userName Returns the current user name setting.

| | | | |
|--------------------------------------|-----------------|----------------|-------------------|
| virtual IString userName() const; | <u>Win</u> Y | <u>PM</u> Y | <u>Motif</u> N |
|--------------------------------------|-----------------|----------------|-------------------|

IDatastoreBase

State Modifiers

These members are used to modify the state of an object.

setAuthentication

Sets the authentication.

| | | | |
|---|------------------------|-----------------------|--------------------------|
| <pre>virtual IDatastoreBase& setAuthentication(const char* aAuthentication);</pre> | <u>Win</u> <i>Y</i> | <u>PM</u> <i>Y</i> | <u>Motif</u> <i>N</i> |
|---|------------------------|-----------------------|--------------------------|

setDatastoreName

Sets the datastore name that is used when a connection is established.

| | | | |
|---|------------------------|-----------------------|--------------------------|
| <pre>virtual IDatastoreBase& setDatastoreName(const char* aDatastoreName);</pre> | <u>Win</u> <i>Y</i> | <u>PM</u> <i>Y</i> | <u>Motif</u> <i>N</i> |
|---|------------------------|-----------------------|--------------------------|

setUserName

Sets the user name.

| | | | |
|---|------------------------|-----------------------|--------------------------|
| <pre>virtual IDatastoreBase& setUserName(const char* aUserName);</pre> | <u>Win</u> <i>Y</i> | <u>PM</u> <i>Y</i> | <u>Motif</u> <i>N</i> |
|---|------------------------|-----------------------|--------------------------|

Transaction Handling

These members are used to commit or to roll back transactions.

commit

Commits a transaction.

| | | | |
|--|------------------------|-----------------------|--------------------------|
| <pre>virtual IDatastoreBase& commit();</pre> | <u>Win</u> <i>Y</i> | <u>PM</u> <i>Y</i> | <u>Motif</u> <i>N</i> |
|--|------------------------|-----------------------|--------------------------|

| Exceptions | |
|----------------------|--|
| IDAConnectionNotOpen | An object of this class is thrown when an operation that requires a connection is attempted before a connection has been established. An example is a call to disconnect before a connection was made. |
| IDAAccessError | An object of this class is thrown when an error occurs during an attempt to access data in the datastore. |

rollback

Performs a rollback on the transactions.

| | | | |
|--|------------------------|-----------------------|--------------------------|
| <pre>virtual IDatastoreBase& rollback();</pre> | <u>Win</u> <i>Y</i> | <u>PM</u> <i>Y</i> | <u>Motif</u> <i>N</i> |
|--|------------------------|-----------------------|--------------------------|

IDatastoreBase

| Exceptions | |
|----------------------|--|
| IDAConnectionNotOpen | An object of this class is thrown when an operation that requires a connection is attempted before a connection has been established. An example is a call to disconnect before a connection was made. |
| IDAAccessError | An object of this class is thrown when an error occurs during an attempt to access data in the datastore. |

Inherited Public Functions

| IStandardNotifier | | |
|---------------------|--------------------|--------------------------|
| disableNotification | enableNotification | isEnabledForNotification |

| INotifier | | |
|---------------------|--------------------|--------------------------|
| disableNotification | enableNotification | isEnabledForNotification |

| IVBase | | |
|-------------|----------|--|
| asDebugInfo | asString | |

| IBase | | |
|-------------|-------------|----------------|
| asDebugInfo | messageFile | setMessageFile |
| asString | messageText | version |

Protected Functions

Constructors

IDatastoreBase has three protected constructors and a destructor.

IDatastoreBase

These protected constructors can only be called from derived classes.

| | | | | |
|----------|-------------------|------------|-----------|--------------|
| 1 | IDatastoreBase(); | <u>Win</u> | <u>PM</u> | <u>Motif</u> |
| | | <i>Y</i> | <i>N</i> | <i>N</i> |

| | | | | |
|----------|---|------------|-----------|--------------|
| 2 | IDatastoreBase(const char* datastoreName); | <u>Win</u> | <u>PM</u> | <u>Motif</u> |
| | | <i>Y</i> | <i>N</i> | <i>N</i> |

IDatastoreBase

| | | | | |
|----------|--|-------------------------------|------------------------------|---------------------------------|
| 3 | <code>IDatastoreBase(const char* datastoreName, const char* userName, const char* authentication);</code> | <u>Win</u> <i>Y</i> | <u>PM</u> <i>N</i> | <u>Motif</u> <i>N</i> |
|----------|--|-------------------------------|------------------------------|---------------------------------|

Inherited Protected Functions

| IStandardNotifier | | |
|-------------------|-----------------|--------------|
| addObserver | notifyObservers | observerList |

| INotifier | | |
|-------------|-----------------|--------------|
| addObserver | notifyObservers | observerList |

Public Data

Notification Event Descriptions

These INotificationId strings are used for all notifications that this class provides to its observers.

AuthenticationId

Notification identifier sent when the authentication attribute is modified.

| | | | |
|---|-------------------------------|------------------------------|---------------------------------|
| <code>static const INotificationId AuthenticationId;</code> | <u>Win</u> <i>Y</i> | <u>PM</u> <i>N</i> | <u>Motif</u> <i>N</i> |
|---|-------------------------------|------------------------------|---------------------------------|

connectedId Notification identifier sent when a connection is established.

| | | | |
|--|-------------------------------|------------------------------|---------------------------------|
| <code>static const INotificationId connectedId;</code> | <u>Win</u> <i>Y</i> | <u>PM</u> <i>N</i> | <u>Motif</u> <i>N</i> |
|--|-------------------------------|------------------------------|---------------------------------|

DatastoreNameId

Notification identifier sent when the *datastoreName* attribute is modified.

| | | | |
|--|-------------------------------|------------------------------|---------------------------------|
| <code>static const INotificationId DatastoreNameId;</code> | <u>Win</u> <i>Y</i> | <u>PM</u> <i>N</i> | <u>Motif</u> <i>N</i> |
|--|-------------------------------|------------------------------|---------------------------------|

disconnectedId

Notification identifier sent when a connection is terminated.

IDatastoreBase

| | | | |
|---|-----------------|----------------|-------------------|
| static const INotificationId disconnectedId; | <u>Win</u> Y | <u>PM</u> N | <u>Motif</u> N |
|---|-----------------|----------------|-------------------|

isConnectedId

Notification identifier sent when the Boolean attribute indicating the connect state changes.

| | | | |
|--|-----------------|----------------|-------------------|
| static const INotificationId isConnectedId; | <u>Win</u> Y | <u>PM</u> N | <u>Motif</u> N |
|--|-----------------|----------------|-------------------|

transactedId Notification identifier sent when a commit or a rollback is completed.

| | | | |
|---|-----------------|----------------|-------------------|
| static const INotificationId transactedId; | <u>Win</u> Y | <u>PM</u> N | <u>Motif</u> N |
|---|-----------------|----------------|-------------------|

UserNameId Notification identifier sent when the *userName* attribute is modified.

| | | | |
|---|-----------------|----------------|-------------------|
| static const INotificationId UserNameId; | <u>Win</u> Y | <u>PM</u> N | <u>Motif</u> N |
|---|-----------------|----------------|-------------------|

Inherited Public Data

| IStandardNotifier | | |
|-------------------|--|--|
| deleteId | | |

Inherited Protected Data

| IBase | | |
|-------------|---------------|--|
| recoverable | unrecoverable | |



IDatastoreDB2

Derivation

```

IBase
IVBase
INotifier
IStandardNotifier
IDatastoreBase
IDatastoreDB2

```

Inherited By None.

Header File

idsmcdb.hpp

Members

| Member | Page | Member | Page |
|------------------|------|-------------------|------|
| Constructor | 625 | isAutoCommit | 627 |
| accessMode | 626 | isolationLevel | 627 |
| accessModeId | 630 | isolationLevelId | 631 |
| autoCommitId | 630 | readyId | 631 |
| commit | 629 | rollback | 629 |
| connect | 625 | setAccessMode | 628 |
| connectString | 627 | setAuthentication | 628 |
| connectStringId | 630 | setConnectString | 628 |
| disconnect | 626 | setDatastoreName | 628 |
| driverPrompt | 627 | setDriverPrompt | 628 |
| driverPromptId | 631 | setIsolationLevel | 628 |
| enableAutoCommit | 627 | setUserName | 628 |
| executeSQL | 625 | ~IDatastoreDB2 | 625 |

Use IDatastoreDB2 with persistent objects that were generated using the CLI option to access DB2 version 1.2 or 2.1 CLI. This class can also be used with embedded SQL. If the user of the application has the authority to use dynamic SQL, the application can be built using IDatastoreDB2, even when generated as embedded SQL. To access DB2 version 1.2 or 2.1 static embedded SQL, use IDatastore (p. 610).



Data Access Builder does not support DB2 version 1.2 on Windows platforms.

Public Functions

Constructors

IDatastoreDB2 has two public constructors and a destructor.

IDatastoreDB2

IDatastoreDB2

This class has two constructors; one takes no parameters, and the other takes one required parameter and one optional parameter.

| | | | | |
|----------|--|-------------------|------------------|---------------------|
| 1 | <code>IDatastoreDB2();</code> | <u>Win</u> | <u>PM</u> | <u>Motif</u> |
| | | <i>Y</i> | <i>N</i> | <i>N</i> |
| 2 | <code>IDatastoreDB2(const char* datastoreName, const char* connectionString = "");</code> | <u>Win</u> | <u>PM</u> | <u>Motif</u> |
| | | <i>Y</i> | <i>N</i> | <i>N</i> |

~IDatastoreDB2

The IDatastoreDB2 destructor frees space allocated by the constructor, and executes a disconnect if the application has not explicitly done so.

| | | | |
|--|-------------------|------------------|---------------------|
| <code>virtual ~IDatastoreDB2();</code> | <u>Win</u> | <u>PM</u> | <u>Motif</u> |
| | <i>Y</i> | <i>N</i> | <i>N</i> |

Data Manipulation

This member is used to execute a dynamic SQL statement.

executeSQL For a description of this function, see IDatastoreBase::executeSQL (p. 617).

Note: Methods implemented in the base class return a reference to the calling class type.

| | | | |
|--|-------------------|------------------|---------------------|
| <code>virtual IDatastoreDB2& executeSQL(const char* aCmd);</code> | <u>Win</u> | <u>PM</u> | <u>Motif</u> |
| | <i>Y</i> | <i>N</i> | <i>N</i> |

Database Connection

These members are used to connect to or to disconnect from a datastore.

connect For a description of this function, see IDatastoreBase::connect (p. 617).

Note: Methods implemented in the base class return a reference to the calling class type.

| | | | | |
|----------|--|-------------------|------------------|---------------------|
| 1 | <code>virtual IDatastoreDB2& connect(const char* datastoreName);</code> | <u>Win</u> | <u>PM</u> | <u>Motif</u> |
| | | <i>Y</i> | <i>N</i> | <i>N</i> |

IDatastoreDB2

| | | | | |
|----------|--|-------------------------------|------------------------------|---------------------------------|
| 2 | <code>virtual IDatastoreDB2& connect();</code> | <u>Win</u> <i>Y</i> | <u>PM</u> <i>Y</i> | <u>Motif</u> <i>N</i> |
|----------|--|-------------------------------|------------------------------|---------------------------------|

| | | | | |
|----------|---|-------------------------------|------------------------------|---------------------------------|
| 3 | <code>virtual IDatastoreDB2& connect(const char* userName, const char* authentication);</code> | <u>Win</u> <i>Y</i> | <u>PM</u> <i>N</i> | <u>Motif</u> <i>N</i> |
|----------|---|-------------------------------|------------------------------|---------------------------------|

| | | | | |
|----------|--|-------------------------------|------------------------------|---------------------------------|
| 4 | <code>virtual IDatastoreDB2& connect(const char* datastoreName, const char* userName, const char* authentication);</code> | <u>Win</u> <i>Y</i> | <u>PM</u> <i>Y</i> | <u>Motif</u> <i>N</i> |
|----------|--|-------------------------------|------------------------------|---------------------------------|

| | | | | |
|----------|--|-------------------------------|------------------------------|---------------------------------|
| 5 | <code>virtual IDatastoreDB2& connect(const char* datastoreName, const char* connectionString, const char* userName, const char* authentication);</code> | <u>Win</u> <i>Y</i> | <u>PM</u> <i>N</i> | <u>Motif</u> <i>N</i> |
|----------|--|-------------------------------|------------------------------|---------------------------------|

| Exceptions | |
|------------------|---|
| IDAConnectFailed | An object of this class is thrown when a database connection attempt fails. |
| IDAAccessError | An object of this class is thrown when an error occurs during an attempt to access data in the datastore. |

disconnect For a description of this function, see IDatastoreBase::disconnect (p. 618).

Note: Methods implemented in the base class return a reference to the calling class type.

| | | | |
|---|-------------------------------|------------------------------|---------------------------------|
| <code>virtual IDatastoreDB2& disconnect();</code> | <u>Win</u> <i>Y</i> | <u>PM</u> <i>Y</i> | <u>Motif</u> <i>N</i> |
|---|-------------------------------|------------------------------|---------------------------------|

Queries

These members are used to access information about an object.

accessMode Returns the current access mode setting.

| | | | |
|--|-------------------------------|------------------------------|---------------------------------|
| <code>virtual IDatastoreDB2::AccessMode accessMode() const;</code> | <u>Win</u> <i>Y</i> | <u>PM</u> <i>N</i> | <u>Motif</u> <i>N</i> |
|--|-------------------------------|------------------------------|---------------------------------|

IDatastoreDB2

connectString

Returns the current connect string setting. This string, which enables a user to specify additional connect string keywords and values, is concatenated with DSN=<value> (*datastoreName*) and, if provided, UID=<value> (*userName*), and PWD=<value> (*authentication*), to form the connect string for the SQLDriverConnect call. For example:

```
"AUTOCOMMIT=0; CONNECTTYPE=1;"
```

```
virtual IString  
    connectString() const;
```

| <u>Win</u> | <u>PM</u> | <u>Motif</u> |
|------------|-----------|--------------|
| <i>Y</i> | <i>N</i> | <i>N</i> |

driverPrompt Returns the current driver prompt parameter setting.

```
virtual IDatastoreDB2::DriverPrompt  
    driverPrompt() const;
```

| <u>Win</u> | <u>PM</u> | <u>Motif</u> |
|------------|-----------|--------------|
| <i>Y</i> | <i>N</i> | <i>N</i> |

isAutoCommit

Returns true if the autocommit mode is enabled; otherwise, returns false.

```
virtual IBoolean  
    isAutoCommit() const;
```

| <u>Win</u> | <u>PM</u> | <u>Motif</u> |
|------------|-----------|--------------|
| <i>Y</i> | <i>N</i> | <i>N</i> |

isolationLevel

Returns the current isolation level setting.

```
virtual IDatastoreDB2::IsolationLevel  
    isolationLevel() const;
```

| <u>Win</u> | <u>PM</u> | <u>Motif</u> |
|------------|-----------|--------------|
| <i>Y</i> | <i>N</i> | <i>N</i> |

State Modifiers

These members are used to modify the state of an object.

enableAutoCommit

Enables or disables the autocommit mode.

```
virtual IDatastoreDB2&  
    enableAutoCommit( IBoolean enable = True);
```

| <u>Win</u> | <u>PM</u> | <u>Motif</u> |
|------------|-----------|--------------|
| <i>Y</i> | <i>N</i> | <i>N</i> |

IDatastoreDB2

setAccessMode

Enables a user to set the access mode.

| | | | |
|--|-------------------------------|------------------------------|---------------------------------|
| <code>virtual IDatastoreDB2& setAccessMode(AccessMode option);</code> | <u>Win</u> <i>Y</i> | <u>PM</u> <i>N</i> | <u>Motif</u> <i>N</i> |
|--|-------------------------------|------------------------------|---------------------------------|

setAuthentication

Sets the authentication.

| | | | |
|--|-------------------------------|------------------------------|---------------------------------|
| <code>virtual IDatastoreDB2& setAuthentication(const char* aAuthentication);</code> | <u>Win</u> <i>Y</i> | <u>PM</u> <i>Y</i> | <u>Motif</u> <i>N</i> |
|--|-------------------------------|------------------------------|---------------------------------|

setConnectionString

Sets the connect string. For a description of the connect string, see
IDatastoreDB2::connectString (p. 627).

| | | | |
|--|-------------------------------|------------------------------|---------------------------------|
| <code>virtual IDatastoreDB2& setConnectionString(const char* aConnectionString);</code> | <u>Win</u> <i>Y</i> | <u>PM</u> <i>N</i> | <u>Motif</u> <i>N</i> |
|--|-------------------------------|------------------------------|---------------------------------|

setDatastoreName

Sets the datastore name that is used when a connection is established.

| | | | |
|--|-------------------------------|------------------------------|---------------------------------|
| <code>virtual IDatastoreDB2& setDatastoreName(const char* aDatastoreName);</code> | <u>Win</u> <i>Y</i> | <u>PM</u> <i>Y</i> | <u>Motif</u> <i>N</i> |
|--|-------------------------------|------------------------------|---------------------------------|

setDriverPrompt

Enables a user to set the driver prompt parameter, and to pass in the window handle
of the parent application.

| | | | |
|---|-------------------------------|------------------------------|---------------------------------|
| <code>virtual IDatastoreDB2& setDriverPrompt(DriverPrompt option, IWindowHandle aHandle = IWindowHandle ());</code> | <u>Win</u> <i>Y</i> | <u>PM</u> <i>N</i> | <u>Motif</u> <i>N</i> |
|---|-------------------------------|------------------------------|---------------------------------|

setIsolationLevel

Enables a user to set the isolation level.

| | | | |
|--|-------------------------------|------------------------------|---------------------------------|
| <code>virtual IDatastoreDB2& setIsolationLevel(IsolationLevel option);</code> | <u>Win</u> <i>Y</i> | <u>PM</u> <i>N</i> | <u>Motif</u> <i>N</i> |
|--|-------------------------------|------------------------------|---------------------------------|

setUserName Sets the user name.

IDatastoreDB2

```
virtual IDatastoreDB2&
    setUsername( const char* aUserName);
```

| <u>Win</u> | <u>PM</u> | <u>Motif</u> |
|------------|-----------|--------------|
| <i>Y</i> | <i>Y</i> | <i>N</i> |

Transaction Handling

These members are used to commit or to roll back transactions.

commit

For a description of this function, see IDatastoreBase::commit (p. 620).

Note: Methods implemented in the base class return a reference to the calling class type.

```
virtual IDatastoreDB2&
    commit();
```

| <u>Win</u> | <u>PM</u> | <u>Motif</u> |
|------------|-----------|--------------|
| <i>Y</i> | <i>Y</i> | <i>N</i> |

rollback

For a description of this function, see IDatastoreBase::rollback (p. 620).

Note: Methods implemented in the base class return a reference to the calling class type.

```
virtual IDatastoreDB2&
    rollback();
```

| <u>Win</u> | <u>PM</u> | <u>Motif</u> |
|------------|-----------|--------------|
| <i>Y</i> | <i>Y</i> | <i>N</i> |

Inherited Public Functions

| IDatastoreBase | | |
|----------------|---------------|-------------------|
| asString | datastoreName | rollback |
| authentication | disconnect | setAuthentication |
| commit | executeSQL | setDatastoreName |
| connect | isConnected | setUserName |

| IStandardNotifier | | |
|---------------------|--------------------|--------------------------|
| disableNotification | enableNotification | isEnabledForNotification |

| INotifier | | |
|---------------------|--------------------|--------------------------|
| disableNotification | enableNotification | isEnabledForNotification |

| IVBase | | |
|-------------|----------|--|
| asDebugInfo | asString | |

IDatastoreDB2

| IBase | | |
|-------------|-------------|----------------|
| asDebugInfo | messageFile | setMessageFile |
| asString | messageText | version |

Inherited Protected Functions

| IStandardNotifier | | |
|-------------------|-----------------|--------------|
| addObserver | notifyObservers | observerList |

| INotifier | | |
|-------------|-----------------|--------------|
| addObserver | notifyObservers | observerList |

Public Data

Notification Event Descriptions

These INotificationId strings are used for all notifications that this class provides to its observers.

accessModeId

Notification identifier sent when the *accessMode* attribute is modified.

| | | | |
|---|-----------------|----------------|-------------------|
| static const INotificationId accessModeId; | <u>Win</u> Y | <u>PM</u> N | <u>Motif</u> N |
|---|-----------------|----------------|-------------------|

autoCommitId

Notification identifier sent when the *autoCommit* attribute is modified.

| | | | |
|---|-----------------|----------------|-------------------|
| static const INotificationId autoCommitId; | <u>Win</u> Y | <u>PM</u> N | <u>Motif</u> N |
|---|-----------------|----------------|-------------------|

connectStringId

Notification identifier sent when the *connectString* attribute is modified.

| | | | |
|--|-----------------|----------------|-------------------|
| static const INotificationId connectStringId; | <u>Win</u> Y | <u>PM</u> N | <u>Motif</u> N |
|--|-----------------|----------------|-------------------|

IDatastoreDB2

driverPromptId

Notification identifier sent when the *driverPrompt* attribute changes.

| | | | |
|------------------------------|------------|-----------|--------------|
| static const INotificationId | <u>Win</u> | <u>PM</u> | <u>Motif</u> |
| driverPromptId; | Y | N | N |

isolationLevelId

Notification identifier sent when the *isolationLevel* attribute changes.

| | | | |
|------------------------------|------------|-----------|--------------|
| static const INotificationId | <u>Win</u> | <u>PM</u> | <u>Motif</u> |
| isolationLevelId; | Y | N | N |

readyId

Notification identifier sent when a part is initialized.

| | | | |
|------------------------------|------------|-----------|--------------|
| static const INotificationId | <u>Win</u> | <u>PM</u> | <u>Motif</u> |
| readyId; | Y | N | N |

Inherited Public Data

| IDatastoreBase | | |
|------------------|-----------------|---------------|
| AuthenticationId | DatastoreNameId | isConnectedId |
| connectedId | disconnectedId | transactedId |

| IStandardNotifier | | |
|-------------------|--|--|
| deleteId | | |

Inherited Protected Data

| IBase | | |
|-------------|---------------|--|
| recoverable | unrecoverable | |



IDatastoreODBC

Derivation

IBase
IVBase
INotifier
IStandardNotifier
IDatastoreBase
IDatastoreODBC

Inherited By None.

Header File

idsmcod.hpp

| Members | Member | Page | Member | Page |
|---------|------------------|------|-------------------|------|
| | Constructor | 632 | isAutoCommit | 635 |
| | accessMode | 634 | isolationLevel | 635 |
| | accessModeId | 638 | isolationLevelId | 639 |
| | autoCommitId | 638 | readyId | 639 |
| | commit | 637 | rollback | 637 |
| | connect | 633 | setAccessMode | 635 |
| | connectString | 634 | setAuthentication | 636 |
| | connectStringId | 638 | setConnectString | 636 |
| | disconnect | 634 | setDatastoreName | 636 |
| | driverPrompt | 635 | setDriverPrompt | 636 |
| | driverPromptId | 639 | setIsolationLevel | 636 |
| | enableAutoCommit | 635 | setUserName | 636 |
| | executeSQL | 633 | ~IDatastoreODBC | 633 |

Use IDatastoreODBC with persistent objects that were generated using the ODBC option to access databases that are managed through ODBC.

Public Functions

Constructors

IDatastoreODBC has two public constructors and a destructor.

IDatastoreODBC

This class has two constructors; one takes no parameters, and the other takes one required parameter and one optional parameter.

IDatastoreODBC

| | | | | |
|----------|-------------------|------------|-----------|--------------|
| 1 | IDatastoreODBC(); | <u>Win</u> | <u>PM</u> | <u>Motif</u> |
| | | <i>Y</i> | <i>N</i> | <i>N</i> |

| | | | | |
|----------|---|------------|-----------|--------------|
| 2 | IDatastoreODBC(const char* datastoreName, const char* connectionString = ""); | <u>Win</u> | <u>PM</u> | <u>Motif</u> |
| | | <i>Y</i> | <i>N</i> | <i>N</i> |

~IDatastoreODBC

The IDatastoreODBC destructor frees space allocated by the constructor, and executes a disconnect if the application has not explicitly done so.

| | | | | |
|---------|--------------------|------------|-----------|--------------|
| virtual | ~IDatastoreODBC(); | <u>Win</u> | <u>PM</u> | <u>Motif</u> |
| | | <i>Y</i> | <i>N</i> | <i>N</i> |

Data Manipulation

This member is used to execute a dynamic SQL statement.

executeSQL For a description of this function, see IDatastoreBase::executeSQL (p. 617).

Note: Methods implemented in the base class return a reference to the calling class type.

| | | | | |
|-------------------------|--------------------------------|------------|-----------|--------------|
| virtual IDatastoreODBC& | executeSQL(const char* aCmd); | <u>Win</u> | <u>PM</u> | <u>Motif</u> |
| | | <i>Y</i> | <i>N</i> | <i>N</i> |

Database Connection

These members are used to connect to or to disconnect from a datastore.

connect For a description of this function, see IDatastoreBase::connect (p. 617).

Note: Methods implemented in the base class return a reference to the calling class type.

| | | | | |
|----------|--------------------------------------|------------|-----------|--------------|
| 1 | virtual IDatastoreODBC& | <u>Win</u> | <u>PM</u> | <u>Motif</u> |
| | connect(const char* datastoreName); | <i>Y</i> | <i>N</i> | <i>N</i> |

| | | | | |
|----------|-------------------------|------------|-----------|--------------|
| 2 | virtual IDatastoreODBC& | <u>Win</u> | <u>PM</u> | <u>Motif</u> |
| | connect(); | <i>Y</i> | <i>Y</i> | <i>N</i> |

IDatastoreODBC

| | | | | |
|----------|---|-------------------------------|------------------------------|---------------------------------|
| 3 | <code>virtual IDatastoreODBC& connect(const char* userName, const char* authentication);</code> | <u>Win</u> <i>Y</i> | <u>PM</u> <i>N</i> | <u>Motif</u> <i>N</i> |
|----------|---|-------------------------------|------------------------------|---------------------------------|

| | | | | |
|----------|---|-------------------------------|------------------------------|---------------------------------|
| 4 | <code>virtual IDatastoreODBC& connect(const char* datastoreName, const char* userName, const char* authentication);</code> | <u>Win</u> <i>Y</i> | <u>PM</u> <i>Y</i> | <u>Motif</u> <i>N</i> |
|----------|---|-------------------------------|------------------------------|---------------------------------|

| | | | | |
|----------|---|-------------------------------|------------------------------|---------------------------------|
| 5 | <code>virtual IDatastoreODBC& connect(const char* datastoreName, const char* connectString, const char* userName, const char* authentication);</code> | <u>Win</u> <i>Y</i> | <u>PM</u> <i>N</i> | <u>Motif</u> <i>N</i> |
|----------|---|-------------------------------|------------------------------|---------------------------------|

| Exceptions | |
|------------------|---|
| IDAConnectFailed | An object of this class is thrown when a database connection attempt fails. |
| IDAAccessError | An object of this class is thrown when an error occurs during an attempt to access data in the datastore. |

disconnect For a description of this function, see IDatastoreBase::disconnect (p. 618).

Note: Methods implemented in the base class return a reference to the calling class type.

| | | | |
|--|-------------------------------|------------------------------|---------------------------------|
| <code>virtual IDatastoreODBC& disconnect();</code> | <u>Win</u> <i>Y</i> | <u>PM</u> <i>Y</i> | <u>Motif</u> <i>N</i> |
|--|-------------------------------|------------------------------|---------------------------------|

Queries

These members are used to access information about an object.

accessMode Returns the current access mode setting.

| | | | |
|---|-------------------------------|------------------------------|---------------------------------|
| <code>virtual IDatastoreODBC::AccessMode accessMode() const;</code> | <u>Win</u> <i>Y</i> | <u>PM</u> <i>N</i> | <u>Motif</u> <i>N</i> |
|---|-------------------------------|------------------------------|---------------------------------|

connectString

Returns the current connect string setting. This string, which enables a user to specify additional connect string keywords and values, is concatenated with DSN=<value> (*datastoreName*) and, if provided, UID=<value> (*userName*), and

IDatastoreODBC

PWD=<value> (*authentication*), to form the connect string for the SQLDriverConnect call. For example:

```
"AUTOCOMMIT=0; CONNECTTYPE=1;"
```

| | | | |
|---|-----------------|----------------|-------------------|
| virtual IString connectString() const; | <u>Win</u> Y | <u>PM</u> N | <u>Motif</u> N |
|---|-----------------|----------------|-------------------|

driverPrompt Returns the current driver prompt parameter setting.

| | | | |
|---|-----------------|----------------|-------------------|
| virtual IDatastoreODBC::DriverPrompt driverPrompt() const; | <u>Win</u> Y | <u>PM</u> N | <u>Motif</u> N |
|---|-----------------|----------------|-------------------|

isAutoCommit

Returns true if the autocommit mode is enabled; otherwise, returns false.

| | | | |
|---|-----------------|----------------|-------------------|
| virtual IBoolean isAutoCommit() const; | <u>Win</u> Y | <u>PM</u> N | <u>Motif</u> N |
|---|-----------------|----------------|-------------------|

isolationLevel

Returns the current isolation level setting.

| | | | |
|---|-----------------|----------------|-------------------|
| virtual IDatastoreODBC::IsolationLevel isolationLevel() const; | <u>Win</u> Y | <u>PM</u> N | <u>Motif</u> N |
|---|-----------------|----------------|-------------------|

State Modifiers

These members are used to modify the state of an object.

enableAutoCommit

Enables or disables the autocommit mode.

| | | | |
|---|-----------------|----------------|-------------------|
| virtual IDatastoreODBC& enableAutoCommit(IBoolean enable = True); | <u>Win</u> Y | <u>PM</u> N | <u>Motif</u> N |
|---|-----------------|----------------|-------------------|

setAccessMode

Enables a user to set the access mode.

| | | | |
|---|-----------------|----------------|-------------------|
| virtual IDatastoreODBC& setAccessMode(AccessMode option); | <u>Win</u> Y | <u>PM</u> N | <u>Motif</u> N |
|---|-----------------|----------------|-------------------|

IDatastoreODBC

setAuthentication

Sets the authentication.

| | | | |
|---|-------------------------------|------------------------------|---------------------------------|
| <code>virtual IDatastoreODBC& setAuthentication(const char* aAuthentication);</code> | <u>Win</u> <i>Y</i> | <u>PM</u> <i>Y</i> | <u>Motif</u> <i>N</i> |
|---|-------------------------------|------------------------------|---------------------------------|

setConnectionString

Sets the connect string. For a description of the connect string, see `IDatastoreODBC::connectString` (p. 634).

| | | | |
|---|-------------------------------|------------------------------|---------------------------------|
| <code>virtual IDatastoreODBC& setConnectionString(const char* aConnectionString);</code> | <u>Win</u> <i>Y</i> | <u>PM</u> <i>N</i> | <u>Motif</u> <i>N</i> |
|---|-------------------------------|------------------------------|---------------------------------|

setDatastoreName

Sets the datastore name that is used when a connection is established.

| | | | |
|---|-------------------------------|------------------------------|---------------------------------|
| <code>virtual IDatastoreODBC& setDatastoreName(const char* aDatastoreName);</code> | <u>Win</u> <i>Y</i> | <u>PM</u> <i>Y</i> | <u>Motif</u> <i>N</i> |
|---|-------------------------------|------------------------------|---------------------------------|

setDriverPrompt

Enables a user to set the driver prompt parameter, and to pass in the window handle of the parent application.

A description of the logon dialog box can be found in the appropriate database section of the supported ODBC drivers chapter of the *User's Guide*.

| | | | |
|--|-------------------------------|------------------------------|---------------------------------|
| <code>virtual IDatastoreODBC& setDriverPrompt(DriverPrompt option, IWindowHandle aHandle = IWindowHandle ());</code> | <u>Win</u> <i>Y</i> | <u>PM</u> <i>N</i> | <u>Motif</u> <i>N</i> |
|--|-------------------------------|------------------------------|---------------------------------|

setIsolationLevel

Enables a user to set the isolation level.

| | | | |
|---|-------------------------------|------------------------------|---------------------------------|
| <code>virtual IDatastoreODBC& setIsolationLevel(IsolationLevel option);</code> | <u>Win</u> <i>Y</i> | <u>PM</u> <i>N</i> | <u>Motif</u> <i>N</i> |
|---|-------------------------------|------------------------------|---------------------------------|

setUserName Sets the user name.

| | | | |
|---|-------------------------------|------------------------------|---------------------------------|
| <code>virtual IDatastoreODBC& setUserName(const char* aUserName);</code> | <u>Win</u> <i>Y</i> | <u>PM</u> <i>Y</i> | <u>Motif</u> <i>N</i> |
|---|-------------------------------|------------------------------|---------------------------------|

IDatastoreODBC

Transaction Handling

These members are used to commit or to roll back transactions.

commit For a description of this function, see IDatastoreBase::commit (p. 620).

Note: Methods implemented in the base class return a reference to the calling class type.

| | | | |
|--------------------------------------|-----------------|----------------|-------------------|
| virtual IDatastoreODBC& commit(); | <u>Win</u> Y | <u>PM</u> Y | <u>Motif</u> N |
|--------------------------------------|-----------------|----------------|-------------------|

rollback For a description of this function, see IDatastoreBase::rollback (p. 620).

Note: Methods implemented in the base class return a reference to the calling class type.

| | | | |
|--|-----------------|----------------|-------------------|
| virtual IDatastoreODBC& rollback(); | <u>Win</u> Y | <u>PM</u> Y | <u>Motif</u> N |
|--|-----------------|----------------|-------------------|

Inherited Public Functions

| IDatastoreBase | | |
|----------------|---------------|-------------------|
| asString | datastoreName | rollback |
| authentication | disconnect | setAuthentication |
| commit | executeSQL | setDatastoreName |
| connect | isConnected | setUserName |

| IStandardNotifier | | |
|---------------------|--------------------|--------------------------|
| disableNotification | enableNotification | isEnabledForNotification |

| INotifier | | |
|---------------------|--------------------|--------------------------|
| disableNotification | enableNotification | isEnabledForNotification |

| IVBase | | |
|-------------|----------|--|
| asDebugInfo | asString | |

| IBase | | |
|-------------|-------------|----------------|
| asDebugInfo | messageFile | setMessageFile |

IDatastoreODBC

| IBase | | |
|----------|-------------|---------|
| asString | messageText | version |

Inherited Protected Functions

| IStandardNotifier | | |
|-------------------|-----------------|--------------|
| addObserver | notifyObservers | observerList |

| INotifier | | |
|-------------|-----------------|--------------|
| addObserver | notifyObservers | observerList |

Public Data

Notification Event Descriptions

These INotificationId strings are used for all notifications that this class provides to its observers.

accessModelId

Notification identifier sent when the *accessMode* attribute is modified.

| | | | |
|---|-----------------|----------------|-------------------|
| static const INotificationId accessModeId; | <u>Win</u> Y | <u>PM</u> N | <u>Motif</u> N |
|---|-----------------|----------------|-------------------|

autoCommitId

Notification identifier sent when the *autoCommit* attribute is modified.

| | | | |
|---|-----------------|----------------|-------------------|
| static const INotificationId autoCommitId; | <u>Win</u> Y | <u>PM</u> N | <u>Motif</u> N |
|---|-----------------|----------------|-------------------|

connectStringId

Notification identifier sent when the *connectString* attribute is modified.

| | | | |
|--|-----------------|----------------|-------------------|
| static const INotificationId connectStringId; | <u>Win</u> Y | <u>PM</u> N | <u>Motif</u> N |
|--|-----------------|----------------|-------------------|

IDatastoreODBC

driverPromptId

Notification identifier sent when the *driverPrompt* attribute changes.

| | | | |
|------------------------------|------------|-----------|--------------|
| static const INotificationId | <u>Win</u> | <u>PM</u> | <u>Motif</u> |
| driverPromptId; | Y | N | N |

isolationLevelId

Notification identifier sent when the *isolationLevel* attribute changes.

| | | | |
|------------------------------|------------|-----------|--------------|
| static const INotificationId | <u>Win</u> | <u>PM</u> | <u>Motif</u> |
| isolationLevelId; | Y | N | N |

readyId

Notification identifier sent when a part is initialized.

| | | | |
|------------------------------|------------|-----------|--------------|
| static const INotificationId | <u>Win</u> | <u>PM</u> | <u>Motif</u> |
| readyId; | Y | N | N |

Inherited Public Data

| IDatastoreBase | | |
|------------------|-----------------|---------------|
| AuthenticationId | DatastoreNameId | isConnectedId |
| connectedId | disconnectedId | transactedId |

| IStandardNotifier | | |
|-------------------|--|--|
| deleteId | | |

Inherited Protected Data

| IBase | | |
|-------------|---------------|--|
| recoverable | unrecoverable | |



Inherited By None.

Header File

ipersist.hpp

| Members | | | | |
|-------------------|------|--------------------|------|--|
| Member | Page | Member | Page | |
| Constructor | 644 | operator = | 644 | |
| add | 641 | operator == | 644 | |
| asString | 642 | retrieve | 641 | |
| del | 641 | setReadOnly | 643 | |
| forDisplay | 642 | setRetrievable | 643 | |
| isDefaultReadOnly | 642 | update | 642 | |
| isReadOnly | 643 | ~IPersistentObject | 640 | |
| isRetrievable | 643 | | | |

The IPersistentObject class provides the basic data manipulation operations that a client can call directly to retrieve, update, add, or delete rows from a table. It is the abstract base class for some of the parts generated by the tool. Objects of this class are not instantiated, but the class may be used as a generic reference or pointer to objects instantiated from the concrete derived classes.

The Data Access Builder generates classes derived from this class. For more information, see the *Open Class Library User's Guide*.

Public Functions

Constructors

IPersistentObject has two protected constructors, a public destructor, and a protected assignment operator.

~IPersistentObject

The IPersistentObject destructor frees space allocated by the constructor.

| | | | |
|-----------------------|------------|-----------|--------------|
| virtual | | | |
| ~IPersistentObject(); | <u>Win</u> | <u>PM</u> | <u>Motif</u> |
| | Y | Y | N |

IPersistentObject

Data Manipulation

These members are used to retrieve, update, add, or delete rows from a table.

add

In the generated part, the derived class adds a row to a table using the data attribute values set in the object. The object should be uniquely identified by the data identifier. **Attention:** If the identifier is not unique, the object may not be retrievable.

```
virtual IPersistentObject&
    add();
```

| <u>Win</u> | <u>PM</u> | <u>Motif</u> |
|------------|-----------|--------------|
| <i>Y</i> | <i>Y</i> | <i>N</i> |

| Exceptions | |
|-----------------------------|---|
| IDAAccessError | An object of this class is thrown when an error occurs during an attempt to access data in the datastore. |
| IDADDataObjectAlreadyExists | An object of this class is thrown when the data object specified in an add operation already exists in the datastore. |

del

In the generated part, the derived class deletes a row from a table using the data identifier set in the object. The composition of the data identifier is defined for the concrete class. **Attention:** If the identifier is not unique, several rows may be deleted. In this case, catch the IDAAccessError exception, and rollback as required.

```
virtual IPersistentObject&
    del();
```

| <u>Win</u> | <u>PM</u> | <u>Motif</u> |
|------------|-----------|--------------|
| <i>Y</i> | <i>Y</i> | <i>N</i> |

| Exceptions | |
|------------------------|---|
| IDAAccessError | An object of this class is thrown when an error occurs during an attempt to access data in the datastore. |
| IDADDataObjectNotFound | An object of this class is thrown when the data object specified in a retrieve, update, or delete operation cannot be found in the datastore. |

retrieve

In the generated part, the derived class retrieves 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 in the object *before* calling retrieve. If the identifier is not unique, this function simply retrieves the first row of the table.

```
virtual IPersistentObject&
    retrieve();
```

| <u>Win</u> | <u>PM</u> | <u>Motif</u> |
|------------|-----------|--------------|
| <i>Y</i> | <i>Y</i> | <i>N</i> |

| Exceptions | |
|----------------|---|
| IDAAccessError | An object of this class is thrown when an error occurs during an attempt to access data in the datastore. |

IPersistentObject

| Exceptions | |
|-----------------------|---|
| IDADataObjectNotFound | An object of this class is thrown when the data object specified in a retrieve, update, or delete operation cannot be found in the datastore. |

update

In the generated part, the derived class updates a row using the data identifier set in the object. The composition of the data identifier is defined by the concrete class.

Attention: If the identifier is not unique, several rows may be updated. In this case, catch the IDAAccessError exception, and rollback as required.

```
virtual IPersistentObject&
    update();
```

| | | |
|------------|-----------|--------------|
| <u>Win</u> | <u>PM</u> | <u>Motif</u> |
| <i>Y</i> | <i>Y</i> | <i>N</i> |

| Exceptions | |
|----------------|---|
| IDAAccessError | An object of this class is thrown when an error occurs during an attempt to access data in the datastore. |

Queries

These members are used to access information about an object.

asString

This function is defined in the generated code. Returns a string containing the concatenated asString representation of each of the attributes of this class. The separator is inserted between each attribute.

```
virtual IString
    asString( const char* separator = ".") const = 0;
```

| | | |
|------------|-----------|--------------|
| <u>Win</u> | <u>PM</u> | <u>Motif</u> |
| <i>Y</i> | <i>N</i> | <i>N</i> |

forDisplay

This function is defined in the generated code. Returns a string containing the concatenated asString representation of each attribute of this class that has been marked in the Data Access Builder to be returned by this method. The separator is inserted between each attribute.

```
virtual IString
    forDisplay( const char* separator = " ") const = 0;
```

| | | |
|------------|-----------|--------------|
| <u>Win</u> | <u>PM</u> | <u>Motif</u> |
| <i>Y</i> | <i>N</i> | <i>N</i> |

isDefaultReadOnly

Returns true if the derived class to which the object belongs is read-only by default; otherwise, returns false. If true:

- An exception occurs following an add, delete, or update statement. You can use only the retrieve statement to read the object from the table.
- You cannot use setReadOnly to change the setting.

IPersistentObject

Boolean Win PM Motif
isDefaultReadOnly() const; Y Y N

isReadOnly Returns true if the table or the object is read-only; otherwise, returns false. If true, an exception occurs when an add, delete, or update method is called.

Boolean Win PM Motif
isReadOnly() const; Y Y N

isRetrievable Returns true if the retrieve method applies to the object; otherwise, returns false.

Boolean Win PM Motif
isRetrievable() const; Y N N

State Modifiers

These members are used to modify the state of an object.

setReadOnly Sets this object to read-only or to updateable.

IPersistentObject& Win PM Motif
setReadOnly(Boolean flag = true); Y Y N

| Exceptions | |
|----------------|---|
| IDAAccessError | An object of this class is thrown when an error occurs during an attempt to access data in the datastore. |

setRetrievable

Specifies whether the retrieve method may be called for this object.

IPersistentObject& Win PM Motif
setRetrievable(Boolean flag = true); Y N N

Inherited Public Functions

| IBase | | |
|-------------|-------------|----------------|
| asDebugInfo | messageFile | setMessageFile |
| asString | messageText | version |

IPersistentObject

Protected Functions

Constructors

IPersistentObject has two protected constructors, a public destructor, and a protected assignment operator.

IPersistentObject

These protected constructors can only be called from derived classes.

| | | | | |
|----------|--|-----------------|----------------|-------------------|
| 1 | IPersistentObject(const Boolean defReadOnly = True); | <u>Win</u> Y | <u>PM</u> Y | <u>Motif</u> N |
| 2 | IPersistentObject(const IPersistentObject& partCopy); | <u>Win</u> Y | <u>PM</u> Y | <u>Motif</u> N |

operator = Assigns one persistent object to another.

| | | | |
|---|-----------------|----------------|-------------------|
| IPersistentObject& operator =(const IPersistentObject& aIPersistentObject); | <u>Win</u> Y | <u>PM</u> Y | <u>Motif</u> N |
|---|-----------------|----------------|-------------------|

Operators

These members compare the attribute values of two objects.

operator == Compares the attribute values of two persistent objects.

| | | | |
|---|-----------------|----------------|-------------------|
| Boolean operator ==(const IPersistentObject& aIPersistentObject) const; | <u>Win</u> Y | <u>PM</u> Y | <u>Motif</u> N |
|---|-----------------|----------------|-------------------|

Inherited Protected Data

| IBase | | |
|-------------|---------------|--|
| recoverable | unrecoverable | |



IPOManager

Derivation Inherits from none.

Inherited By None.

Header File
ipersist.hpp

| Members | Member | Page | Member | Page |
|---------|-------------|------|-------------|------|
| | Constructor | 646 | select | 646 |
| | operator = | 646 | ~IPOManager | 645 |
| | refresh | 645 | | |

The IPOManager class provides operations for the retrieval of a set of results from a table. These results are kept in a collection using the IOC collection classes. Objects of the IPOManager class are not instantiated, but the class may be used as a generic reference or pointer to objects instantiated from the concrete derived classes.

For more information, see the *Open Class Library User's Guide*.

Public Functions

Constructors

IPOManager has two protected constructors, a public destructor, and a protected assignment operator.

~IPOManager

The IPOManager destructor frees space allocated by the constructor.

```
virtual  
~IPOManager();
```

| | | |
|------------|-----------|--------------|
| <u>Win</u> | <u>PM</u> | <u>Motif</u> |
| <i>Y</i> | <i>Y</i> | <i>N</i> |

Data Manipulation

These members are used to retrieve rows from a table.

refresh Retrieves all the rows from a table.

IPOManager

```
virtual IPOManager&
refresh() = 0;
```

| <u>Win</u> | <u>PM</u> | <u>Motif</u> |
|------------|-----------|--------------|
| <i>Y</i> | <i>Y</i> | <i>N</i> |

select

In the generated part, the derived class retrieves all the rows from a table that match the criteria specified by the clause string, an SQL **where** clause. The keyword is added by the class library code. The clause may be followed by additional parameters, as permitted in select clauses by the datastore.

```
virtual IPOManager&
select( const char* clause) = 0;
```

| <u>Win</u> | <u>PM</u> | <u>Motif</u> |
|------------|-----------|--------------|
| <i>Y</i> | <i>Y</i> | <i>N</i> |

Protected Functions

Constructors

IPOManager has two protected constructors, a public destructor, and a protected assignment operator.

IPOManager These protected constructors can only be called from derived classes.

1 IPOManager();

| <u>Win</u> | <u>PM</u> | <u>Motif</u> |
|------------|-----------|--------------|
| <i>Y</i> | <i>Y</i> | <i>N</i> |

2 IPOManager(const IPOManager& partCopy);

| <u>Win</u> | <u>PM</u> | <u>Motif</u> |
|------------|-----------|--------------|
| <i>Y</i> | <i>Y</i> | <i>N</i> |

operator = Assigns one persistent object manager to another.

```
IPOManager&
operator =( const IPOManager& aIPOManager);
```

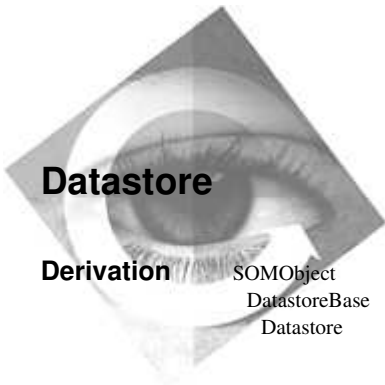
| <u>Win</u> | <u>PM</u> | <u>Motif</u> |
|------------|-----------|--------------|
| <i>Y</i> | <i>N</i> | <i>N</i> |

Part 9. Data Access Builder SOM Classes and Exceptions

Use the database access classes to connect to and to disconnect from your DB2 and other supported databases, and to apply transactions against those databases.

This section describes the SOM version of the database access classes. Use these classes when you want to use SOM objects.

| | |
|---|-----|
| Datastore | 648 |
| Public Functions | 648 |
| Inherited Public Functions | 649 |
| DatastoreBase | 650 |
| Public Functions | 651 |
| DatastoreDB2 | 655 |
| Public Functions | 655 |
| Inherited Public Functions | 658 |
| DatastoreDB2Factory | 660 |
| Public Functions | 660 |
| DatastoreFactory | 661 |
| Public Functions | 661 |
| DatastoreODBC | 662 |
| Public Functions | 662 |
| Inherited Public Functions | 665 |
| DatastoreODBCFactory | 667 |
| Public Functions | 667 |
| PersistentObject | 668 |
| Public Functions | 668 |
| POFactory | 670 |
| Public Functions | 670 |



Datastore

Derivation SOMObject
 DatastoreBase
 Datastore

Inherited By None.

Header File
 sdsmcon.idl

| Members | Member | Page |
|---------|----------------------------|------|
| | enable_sharemode_exclusive | 649 |
| | is_sharemode_exclusive | 648 |

Use Datastore with persistent objects that were generated using the embedded SQL option to access DB2 version 1.2 or 2.1 embedded SQL. Data Access Builder permits only one Datastore instance connection to DB2 per application process. To access DB2 version 2.1 or later using CLI, use DatastoreDB2 (p. 655).

Only one connection should be active at a time when using embedded SQL.

A Datastore can be used with a DatastoreODBC, and a DatastoreDB2 can be used with a DatastoreODBC, but a Datastore *cannot* be used with a DatastoreDB2.

For DB2 version 1.2 and 2.1 embedded SQL, if *user_name* and *authentication* are both specified, these attributes are used to request access to the database; otherwise, an attempt is made to access the database without this information.

 Data Access Builder does not support DB2 version 1.2 on Windows platforms.

Public Functions

Queries

These methods are used to access information about an object.

is_sharemode_exclusive

Returns true if the share mode has been enabled, or false if it has been disabled for this connection.

Datastore

```
boolean  
    is_sharemode_exclusive();
```

| <u>Win</u> | <u>PM</u> | <u>Motif</u> |
|------------|-----------|--------------|
| <i>Y</i> | <i>Y</i> | <i>N</i> |

State Modifiers

These methods are used to modify the state of an object.

enable_sharemode_exclusive

Enables or disables the share mode in a database connection. Enabling this option specifies exclusive access to the database; no other process will be able to connect to the database until the original connection is reset.

```
void  
    enable_sharemode_exclusive( in boolean enable);
```

| <u>Win</u> | <u>PM</u> | <u>Motif</u> |
|------------|-----------|--------------|
| <i>Y</i> | <i>Y</i> | <i>N</i> |

Inherited Public Functions

| DatastoreBase | | |
|-----------------------|---------------------|---------------------|
| commit | disconnect | _get_datastore_name |
| connect | executeSQL | _get_user_name |
| connect_datastorename | is_connected | _set_authentication |
| connect_defaults | rollback | _set_datastore_name |
| connect_user | _get_authentication | _set_user_name |



DatastoreBase

Derivation SOMObject
 DatastoreBase

Inherited By Datastore
 DatastoreDB2
 DatastoreODBC

Header File
 sdsmcbs.idl

| Members | | Member | Page | Member | Page |
|---------|--|-----------------------|------|---------------------|------|
| | | commit | 654 | rollback | 654 |
| | | connect | 651 | _get_authentication | 653 |
| | | connect_datastorename | 651 | _get_datastore_name | 653 |
| | | connect_defaults | 652 | _get_user_name | 653 |
| | | connect_user | 652 | _set_authentication | 653 |
| | | disconnect | 652 | _set_datastore_name | 653 |
| | | executeSQL | 651 | _set_user_name | 653 |
| | | is_connected | 652 | | |

DatastoreBase is an abstract base class for all Data Access Builder SOM classes that represent datastore connections. DatastoreBase provides client connection to the database, disconnection from the database, and commit and rollback of transactions.

The following attributes are used by DatastoreBase:

- authentication
Authentication is the password for the user identified by *user_name*. An unspecified value is indicated by " ".
- datastore_name
Name of the datastore to which the user wishes to connect. In DB2, *datastore_name* represents the name of the database; in ODBC, it represents the name of the datasource.
- user_name
Name of the user to be passed to the datasource. This name and the password are used to complete authorization of the user. If unspecified (or set to " "), the *user_name* is not passed to the datasource, and a connection attempt may fail, or the datasource may attempt to determine the user's authorization independently.

Public Functions

Data Manipulation

This method is used to execute a dynamic SQL statement.

executeSQL Executes a dynamic SQL statement. If a **select** statement is passed in:

- Datastore always returns an exception
- DatastoreDB2 and DatastoreODBC ignore any result other than an access error message.

```
void                                     Win PM Motif
executeSQL( in string command);         Y   N   N
```

| Exceptions | |
|----------------------|---|
| DatastoreAccessError | Set when an error occurs during an attempt to access data in the datastore. |

Database Connection

These methods are used to connect to or to disconnect from a datastore.

connect Connects to a datastore using the specified input parameters. If a connection already exists, performs a disconnect, and then reconnects using the current settings. The values of *user_name* and *authentication* are not saved.

```
void                                     Win PM Motif
connect( in string datastore_name,      Y   Y   N
         in string user_name,
         in string authentication);
```

| Exceptions | |
|-----------------------|---|
| ConnectFailed | Set when a database connection attempt fails. |
| DatastoreAccessError | Set when an error occurs during an attempt to access data in the datastore. |
| DatastoreLogoffFailed | Set when a logoff fails. |
| DatastoreLogonFailed | Set when a logon attempt fails. |

connect_datastorename

Connects to a datastore using the specified *datastore_name* and the current user ID and authentication settings.

DatastoreBase

```
void
    connect_datastorename( in string datastore_name);
```

| <u>Win</u> | <u>PM</u> | <u>Motif</u> |
|------------|-----------|--------------|
| <i>Y</i> | <i>N</i> | <i>N</i> |

connect_defaults

Connects to a datastore using the currently specified datastore settings.

```
void
    connect_defaults();
```

| <u>Win</u> | <u>PM</u> | <u>Motif</u> |
|------------|-----------|--------------|
| <i>Y</i> | <i>Y</i> | <i>N</i> |

connect_user

Connects to a datastore using the specified input parameters and the current *datastore_name* setting. The values of *user_name* and *authentication* are not saved.

```
void
    connect_user( in string user_name,
                  in string authentication);
```

| <u>Win</u> | <u>PM</u> | <u>Motif</u> |
|------------|-----------|--------------|
| <i>Y</i> | <i>N</i> | <i>N</i> |

disconnect Closes the connection to a database. If a logon was performed on the connect, *user_name* is logged off.

```
void
    disconnect();
```

| <u>Win</u> | <u>PM</u> | <u>Motif</u> |
|------------|-----------|--------------|
| <i>Y</i> | <i>Y</i> | <i>N</i> |

| Exceptions | |
|----------------------------|--|
| DatastoreConnectionNotOpen | Set when an operation that requires a connection is attempted before a connection has been established. An example is a call to disconnect before a connection was made. |
| DatastoreLogoffFailed | Set when a logoff fails. |
| DisconnectError | Set when a disconnection error occurs. |

Queries

These methods are used to access information about an object.

is_connected

Returns true if a connection to the database exists.

```
boolean
    is_connected();
```

| <u>Win</u> | <u>PM</u> | <u>Motif</u> |
|------------|-----------|--------------|
| <i>Y</i> | <i>Y</i> | <i>N</i> |

DatastoreBase

`_get_authentication`

Returns the current authentication setting.

`_get_authentication();`

| <u>Win</u> | <u>PM</u> | <u>Motif</u> |
|------------|-----------|--------------|
| <i>Y</i> | <i>Y</i> | <i>N</i> |

`_get_datastore_name`

Returns the current datastore name setting.

`_get_datastore_name();`

| <u>Win</u> | <u>PM</u> | <u>Motif</u> |
|------------|-----------|--------------|
| <i>Y</i> | <i>Y</i> | <i>N</i> |

`_get_user_name`

Returns the current user name setting.

`_get_user_name();`

| <u>Win</u> | <u>PM</u> | <u>Motif</u> |
|------------|-----------|--------------|
| <i>Y</i> | <i>Y</i> | <i>N</i> |

State Modifiers

These methods are used to modify the state of an object.

`_set_authentication`

Sets the authentication.

`_set_authentication();`

| <u>Win</u> | <u>PM</u> | <u>Motif</u> |
|------------|-----------|--------------|
| <i>Y</i> | <i>Y</i> | <i>N</i> |

`_set_datastore_name`

Sets the datastore name that is used when a connection is established.

`_set_datastore_name();`

| <u>Win</u> | <u>PM</u> | <u>Motif</u> |
|------------|-----------|--------------|
| <i>Y</i> | <i>Y</i> | <i>N</i> |

`_set_user_name`

Sets the user name.

`_set_user_name();`

| <u>Win</u> | <u>PM</u> | <u>Motif</u> |
|------------|-----------|--------------|
| <i>Y</i> | <i>Y</i> | <i>N</i> |

DatastoreBase

Transaction Handling

These methods are used to commit or to roll back transactions.

commit Commits a transaction.

```
void  
commit();
```

| <u>Win</u> | <u>PM</u> | <u>Motif</u> |
|------------|-----------|--------------|
| <i>Y</i> | <i>Y</i> | <i>N</i> |

| Exceptions | |
|----------------------------|--|
| DatastoreConnectionNotOpen | Set when an operation that requires a connection is attempted before a connection has been established. An example is a call to disconnect before a connection was made. |
| DatastoreAccessError | Set when an error occurs during an attempt to access data in the datastore. |

rollback Performs a rollback on the transactions.

```
void  
rollback();
```

| <u>Win</u> | <u>PM</u> | <u>Motif</u> |
|------------|-----------|--------------|
| <i>Y</i> | <i>Y</i> | <i>N</i> |

| Exceptions | |
|----------------------------|--|
| DatastoreConnectionNotOpen | Set when an operation that requires a connection is attempted before a connection has been established. An example is a call to disconnect before a connection was made. |
| DatastoreAccessError | Set when an error occurs during an attempt to access data in the datastore. |



DatastoreDB2

Derivation SOMObject
 DatastoreBase
 DatastoreDB2

Inherited By None.

Header File
 sdsmcdb.idl

| Members | Member | Page | Member | Page |
|---------|--------------------|------|----------------------|------|
| | connect_string | 655 | _get_auto_commit | 656 |
| | get_connect_string | 656 | _get_isolation_level | 657 |
| | get_driver_prompt | 656 | _set_access_mode | 657 |
| | set_connect_string | 657 | _set_auto_commit | 658 |
| | set_driver_prompt | 657 | _set_isolation_level | 658 |
| | _get_access_mode | 656 | | |

Use DatastoreDB2 with persistent objects that were generated using the CLI option to access DB2 version 1.2 or 2.1 CLI. This class can also be used with embedded SQL. If the user of the application has the authority to use dynamic SQL, the application can be built using IDatastoreDB2, even when generated as embedded SQL. To access DB2 version 1.2 or 2.1 static embedded SQL, use Datastore (p. 648).

 Data Access Builder does not support DB2 version 1.2 on Windows platforms.

Public Functions

Database Connection

These methods are used to connect to a datastore.

connect_string

Connects to a datastore using the specified input parameters. If a connection already exists, performs a disconnect, and then reconnects using the current settings. The values of *user_name* and *authentication* are not saved.

DatastoreDB2

```
void                                     Win PM Motif  
    connect_string( in string datastore_name,      Y   N   N  
                   in string connect_stringi,  
                   in string user_name,  
                   in string authentication);
```

| Exceptions | |
|----------------------|---|
| ConnectFailed | Set when a database connection attempt fails. |
| DatastoreAccessError | Set when an error occurs during an attempt to access data in the datastore. |

Queries

These methods are used to access information about an object.

get_connect_string

Returns the current connect string setting. This string, which enables a user to specify additional connect string keywords and values, is concatenated with DSN=<value> (*datastore_name*), UID=<value> (*user_name*), and PWD=<value> (*authentication*), to form the connect string for the SQLDriverConnect call. For example:

```
"AUTOCOMMIT=0; CONNECTTYPE=1;"
```

```
string                                     Win PM Motif  
    get_connect_string();                 Y   N   N
```

get_driver_prompt

Returns the current driver prompt parameter setting.

```
long                                     Win PM Motif  
    get_driver_prompt();                 Y   N   N
```

_get_access_mode

Returns the current access mode setting.

```
_get_access_mode();                     Win PM Motif  
                                         Y   N   N
```

_get_auto_commit

Returns true if the autocommit mode is enabled; otherwise, returns false.

DatastoreDB2

`_get_auto_commit();`

| <u>Win</u> | <u>PM</u> | <u>Motif</u> |
|------------|-----------|--------------|
| <i>Y</i> | <i>N</i> | <i>N</i> |

`_get_isolation_level`

Returns the current isolation level setting.

`_get_isolation_level();`

| <u>Win</u> | <u>PM</u> | <u>Motif</u> |
|------------|-----------|--------------|
| <i>Y</i> | <i>N</i> | <i>N</i> |

State Modifiers

These methods are used to modify the state of an object.

`set_connect_string`

Sets the connect string.

`void
set_connect_string(in string connect_string);`

| <u>Win</u> | <u>PM</u> | <u>Motif</u> |
|------------|-----------|--------------|
| <i>Y</i> | <i>N</i> | <i>N</i> |

`set_driver_prompt`

Enables a user to set the driver prompt parameter, and to pass in a window handle.

Valid values are:

- **DAX_DRIVER_PROMPT**
Dialog enabling the user to enter information needed to logon is always initiated.
- **DAX_DRIVER_COMPLETE**
Dialog is only initiated if there is insufficient information in the connection string.
- **DAX_DRIVER_COMPLETE_REQ**
Dialog is only initiated if there is insufficient information in the connection string.
Only mandatory information is requested.
- **DAX_DRIVER_NOPROMPT (Default)**
The user is not prompted for any information.

`void
set_driver_prompt(in long option,
 in void* hwnd);`

| <u>Win</u> | <u>PM</u> | <u>Motif</u> |
|------------|-----------|--------------|
| <i>Y</i> | <i>N</i> | <i>N</i> |

`_set_access_mode`

Enables a user to set the access mode. Valid values are:

- **DAX_READONLY**

DatastoreDB2

Access mode is read-only.

- DAX_READWRITE (Default)
Access mode is updateable.

`_set_access_mode();`

| <u>Win</u> | <u>PM</u> | <u>Motif</u> |
|------------|-----------|--------------|
| <i>Y</i> | <i>N</i> | <i>N</i> |

`_set_auto_commit`

Enables or disables the autocommit mode. Valid values are:

- DAX_AUTOCOMMIT_ON
- DAX_AUTOCOMMIT_OFF (Default)

`_set_auto_commit();`

| <u>Win</u> | <u>PM</u> | <u>Motif</u> |
|------------|-----------|--------------|
| <i>Y</i> | <i>N</i> | <i>N</i> |

`_set_isolation_level`

Enables a user to set the isolation level. Valid values are:

- DAX_READ_UNCOMMITTED
Permits an application to access uncommitted changes caused by other transactions.
- DAX_READ_COMMITTED
Ensures that the current row of every updateable cursor is not changed by other application processes.
- DAX_REPEATABLE_READ
Keeps a lock on all rows accessed since the last commit point.
- DAX_SERIALIZABLE
Data affected by pending transactions is not available to other transactions.
- DAX_DATABASE_DEFAULT (Default)
The default isolation level for the specified database product.

`_set_isolation_level();`

| <u>Win</u> | <u>PM</u> | <u>Motif</u> |
|------------|-----------|--------------|
| <i>Y</i> | <i>N</i> | <i>N</i> |

Inherited Public Functions

DatastoreDB2

| DatastoreBase | | |
|-----------------------|---------------------|---------------------|
| commit | disconnect | _get_datastore_name |
| connect | executeSQL | _get_user_name |
| connect_datastorename | is_connected | _set_authentication |
| connect_defaults | rollback | _set_datastore_name |
| connect_user | _get_authentication | _set_user_name |



DatastoreDB2Factory

Derivation SOMObject
 SOMClass
 DatastoreDB2Factory

Inherited By None.

Header File
 sdsmcdb.idl

| Members | Member | Page |
|---------|------------------------|------|
| | create_object | 660 |
| | create_object_defaults | 660 |

DatastoreDB2Factory is the metaclass for the DatastoreDB2 class; see DatastoreDB2 (p. 655).

Public Functions

Constructors

These methods are used to create a DatastoreDB2 object.

create_object

Use this method to create a DatastoreDB2 object. Initializes *datastore_name* and *connect_string* to empty strings.

| | | | |
|------------------|------------|-----------|--------------|
| DatastoreDB2 | <u>Win</u> | <u>PM</u> | <u>Motif</u> |
| create_object(); | Y | N | N |

create_object_defaults

Use this method to create a DatastoreDB2 object. Initializes *datastore_name* and *connect_string* to the values specified in the call.

| | | | |
|---|------------|-----------|--------------|
| DatastoreDB2 | <u>Win</u> | <u>PM</u> | <u>Motif</u> |
| create_object_defaults(in string datastore_name, | Y | N | N |
| in string connect_string); | | | |



DatastoreFactory

Derivation SOMObject
 SOMClass
 DatastoreFactory

Inherited By None.

Header File
 sdscon.idl

| Members | Member | Page |
|----------------|------------------------|-------------|
| | create_object | 661 |
| | create_object_defaults | 661 |

DatastoreFactory is the metaclass for the Datastore class; see Datastore (p. 648).

Public Functions

Constructors

These methods are used to create a Datastore object.

create_object

Use this method to create a Datastore object. Initializes *datastore_name*, *user_name*, and *authentication* to empty strings.

| | | | | |
|------------------|--|-------------------|------------------|---------------------|
| Datastore | | <u>Win</u> | <u>PM</u> | <u>Motif</u> |
| create_object(); | | <i>Y</i> | <i>Y</i> | <i>N</i> |

create_object_defaults

Use this method to create a Datastore object. Initializes *datastore_name*, *user_name*, and *authentication* to the values specified in the call.

| | | | | |
|-------------------------|----------------------------|-------------------|------------------|---------------------|
| Datastore | | <u>Win</u> | <u>PM</u> | <u>Motif</u> |
| create_object_defaults(| in string datastore_name, | <i>Y</i> | <i>Y</i> | <i>N</i> |
| | in string user_name, | | | |
| | in string authentication); | | | |



Derivation SOMObject
 DatastoreBase
 DatastoreODBC

Inherited By None.

Header File
 sdsmcod.idl

| Members | Member | Page | Member | Page |
|---------|--------------------|------|----------------------|------|
| | connect_string | 662 | _get_auto_commit | 663 |
| | get_connect_string | 663 | _get_isolation_level | 663 |
| | get_driver_prompt | 663 | _set_access_mode | 664 |
| | set_connect_string | 664 | _set_auto_commit | 665 |
| | set_driver_prompt | 664 | _set_isolation_level | 665 |
| | _get_access_mode | 663 | | |

Use DatastoreODBC with persistent objects that were generated using the ODBC option to access databases that are managed through ODBC.

Public Functions

Database Connection

These methods are used to connect to a datastore.

connect_string

Connects to a datastore using the specified input parameters. If a connection already exists, performs a disconnect, and then reconnects using the current settings. The values of *user_name* and *authentication* are not saved.

```
void
  connect_string( in string datastore_name,
                 in string connect_stringi,
                 in string user_name,
                 in string authentication);
```

| | | |
|------------|-----------|--------------|
| Win | PM | Motif |
| Y | N | N |

| Exceptions | |
|---------------|---|
| ConnectFailed | Set when a database connection attempt fails. |

DatastoreODBC

| Exceptions | |
|----------------------|---|
| DatastoreAccessError | Set when an error occurs during an attempt to access data in the datastore. |

Queries

These methods are used to access information about an object.

get_connect_string

Returns the current connect string setting. This string, which enables a user to specify additional connect string keywords and values, is concatenated with DSN=<value> (*datastore_name*), UID=<value> (*user_name*), and PWD=<value> (*authentication*), to form the connect string for the SQLDriverConnect call. For example:

```
"AUTOCOMMIT=0; CONNECTTYPE=1;"
```

| | | | |
|-----------------------|-------------------|------------------|---------------------|
| string | <u>Win</u> | <u>PM</u> | <u>Motif</u> |
| get_connect_string(); | <i>Y</i> | <i>N</i> | <i>N</i> |

get_driver_prompt

Returns the current driver prompt parameter setting.

| | | | |
|----------------------|-------------------|------------------|---------------------|
| long | <u>Win</u> | <u>PM</u> | <u>Motif</u> |
| get_driver_prompt(); | <i>Y</i> | <i>N</i> | <i>N</i> |

_get_access_mode

Returns the current access mode setting.

| | | | |
|---------------------|-------------------|------------------|---------------------|
| _get_access_mode(); | <u>Win</u> | <u>PM</u> | <u>Motif</u> |
| | <i>Y</i> | <i>N</i> | <i>N</i> |

_get_auto_commit

Returns true if the autocommit mode is enabled; otherwise, returns false.

| | | | |
|---------------------|-------------------|------------------|---------------------|
| _get_auto_commit(); | <u>Win</u> | <u>PM</u> | <u>Motif</u> |
| | <i>Y</i> | <i>N</i> | <i>N</i> |

_get_isolation_level

Returns the current isolation level setting.

DatastoreODBC

```
_get_isolation_level();
```

| <u>Win</u> | <u>PM</u> | <u>Motif</u> |
|------------|-----------|--------------|
| <i>Y</i> | <i>N</i> | <i>N</i> |

State Modifiers

These methods are used to modify the state of an object.

set_connect_string

Sets the connect string.

```
void  
set_connect_string( in string connect_string);
```

| <u>Win</u> | <u>PM</u> | <u>Motif</u> |
|------------|-----------|--------------|
| <i>Y</i> | <i>N</i> | <i>N</i> |

set_driver_prompt

Enables a user to set the driver prompt parameter, and to pass in a window handle.

Valid values are:

- DAX_DRIVER_PROMPT
Dialog enabling the user to enter information needed to logon is always initiated.
- DAX_DRIVER_COMPLETE
Dialog is only initiated if there is insufficient information in the connection string.
- DAX_DRIVER_COMPLETE_REQ
Dialog is only initiated if there is insufficient information in the connection string.
Only mandatory information is requested.
- DAX_DRIVER_NOPROMPT (Default)
The user is not prompted for any information.

A description of the logon dialog box can be found in the appropriate database section of the supported ODBC drivers chapter of the *User's Guide*.

```
void  
set_driver_prompt( in long option,  
                  in void* hwnd);
```

| <u>Win</u> | <u>PM</u> | <u>Motif</u> |
|------------|-----------|--------------|
| <i>Y</i> | <i>N</i> | <i>N</i> |

_set_access_mode

Enables a user to set the access mode. Valid values are:

- DAX_READONLY
Access mode is read-only.
- DAX_READWRITE (Default)
Access mode is updateable.

DatastoreODBC

`_set_access_mode();`

| <u>Win</u> | <u>PM</u> | <u>Motif</u> |
|------------|-----------|--------------|
| <i>Y</i> | <i>N</i> | <i>N</i> |

`_set_auto_commit`

Enables or disables the autocommit mode. Valid values are:

- DAX_AUTOCOMMIT_ON
- DAX_AUTOCOMMIT_OFF (Default)

`_set_auto_commit();`

| <u>Win</u> | <u>PM</u> | <u>Motif</u> |
|------------|-----------|--------------|
| <i>Y</i> | <i>N</i> | <i>N</i> |

`_set_isolation_level`

Enables a user to set the isolation level. Valid values are:

- DAX_READ_UNCOMMITTED
Permits an application to access uncommitted changes caused by other transactions.
- DAX_READ_COMMITTED
Ensures that the current row of every updateable cursor is not changed by other application processes.
- DAX_REPEATABLE_READ
Keeps a lock on all rows accessed since the last commit point.
- DAX_SERIALIZABLE
Data affected by pending transactions is not available to other transactions.
- DAX_DATABASE_DEFAULT (Default)
The default isolation level for the specified database product.

`_set_isolation_level();`

| <u>Win</u> | <u>PM</u> | <u>Motif</u> |
|------------|-----------|--------------|
| <i>Y</i> | <i>N</i> | <i>N</i> |

Inherited Public Functions

| DatastoreBase | | |
|-----------------------|--------------|---------------------|
| commit | disconnect | _get_datastore_name |
| connect | executeSQL | _get_user_name |
| connect_datastorename | is_connected | _set_authentication |

DatastoreODBC

| DatastoreBase | | |
|------------------|---------------------|---------------------|
| connect_defaults | rollback | _set_datastore_name |
| connect_user | _get_authentication | _set_user_name |



DatastoreODBCFactory

Derivation SOMObject
 SOMClass
 DatastoreODBCFactory

Inherited By None.

Header File
 sdsmcod.idl

| Members | Member | Page |
|----------------|------------------------|-------------|
| | create_object | 667 |
| | create_object_defaults | 667 |

DatastoreODBCFactory is the metaclass for the DatastoreODBC class; see DatastoreODBC (p. 662).

Public Functions

Constructors

These methods are used to create a DatastoreODBC object.

create_object

Use this method to create a DatastoreODBC object. Initializes *datastore_name* and *connect_string* to empty strings.

| | | | |
|------------------|-------------------|------------------|---------------------|
| DatastoreODBC | <u>Win</u> | <u>PM</u> | <u>Motif</u> |
| create_object(); | <i>Y</i> | <i>N</i> | <i>N</i> |

create_object_defaults

Use this method to create a DatastoreODBC object. Initializes *datastore_name* and *connect_string* to the values specified in the call.

| | | | |
|---|-------------------|------------------|---------------------|
| DatastoreODBC | <u>Win</u> | <u>PM</u> | <u>Motif</u> |
| create_object_defaults(in string datastore_name, | <i>Y</i> | <i>N</i> | <i>N</i> |
| in string connect_string); | | | |



PersistentObject

Derivation SOMObject
PersistentObject

Inherited By None.

Header File
spersist.idl

| Members | | Member | Page | Member | Page |
|---------|--|--------|------|----------|------|
| | | add | 668 | retrieve | 669 |
| | | del | 669 | update | 669 |

The PersistentObject class provides the basic data manipulation operations that a client can call directly to retrieve, update, add, or delete rows from a table. It is the abstract base class for some of the parts generated by the tool.

The Data Access Builder generates classes derived from this class. For more information, see the *Open Class Library User's Guide*.

Public Functions

Data Manipulation

These methods are used to retrieve, update, add, or delete rows from a table.

add In the generated part, the derived class adds a row to a table using the data attribute values set in the object. The object should be uniquely identified by the data identifier. **Attention:** If the identifier is not unique, the object may not be retrievable.

| | | | |
|--------|------------|-----------|--------------|
| void | <u>Win</u> | <u>PM</u> | <u>Motif</u> |
| add(); | Y | Y | N |

| Exceptions | |
|-------------------------|---|
| DataObjectAlreadyExists | Set when the data object specified in an add operation already exists in the datastore. |

PersistentObject

del

In the generated part, the derived class deletes a row from a table using the data identifier set in the object. The composition of the data identifier is defined for the concrete class. **Attention:** If the identifier is not unique, several rows may be deleted. In this case, catch the DatastoreAccessError exception, and rollback as required.

```
void
del();
```

| <u>Win</u> | <u>PM</u> | <u>Motif</u> |
|------------|-----------|--------------|
| <i>Y</i> | <i>Y</i> | <i>N</i> |

| Exceptions | |
|--------------------|---|
| DataObjectNotFound | Set when the data object specified in a retrieve, update, or delete operation cannot be found in the datastore. |

retrieve

In the generated part, the derived class retrieves 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 in the object *before* calling retrieve. If the identifier is not unique, this function simply retrieves the first row of the table.

```
void
retrieve();
```

| <u>Win</u> | <u>PM</u> | <u>Motif</u> |
|------------|-----------|--------------|
| <i>Y</i> | <i>Y</i> | <i>N</i> |

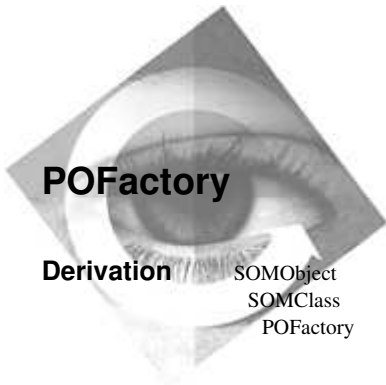
| Exceptions | |
|--------------------|---|
| DataObjectNotFound | Set when the data object specified in a retrieve, update, or delete operation cannot be found in the datastore. |

update

In the generated part, the derived class updates a row in a table using the data identifier set in the object. The composition of the data identifier is defined by the concrete class. **Attention:** If the identifier is not unique, several rows may be updated. In this case, catch the DatastoreAccessError exception, and rollback as required.

```
void
update();
```

| <u>Win</u> | <u>PM</u> | <u>Motif</u> |
|------------|-----------|--------------|
| <i>Y</i> | <i>Y</i> | <i>N</i> |



Inherited By None.

Header File
spersist.idl

| Members | Member | Page |
|---------|-------------|------|
| | retrieveAll | 670 |
| | select | 670 |

The POFactory class provides operations to deal with collections of rows from a table. It is the metaclass for the PersistentObject class; see PersistentObject (p. 668).

Public Functions

Data Manipulation

These methods are used to retrieve rows from a table.

retrieveAll Retrieves all the rows from a table.

| | | | |
|---|-----------------|----------------|-------------------|
| sequence < PersistentObject > retrieveAll(); | <u>Win</u> Y | <u>PM</u> Y | <u>Motif</u> N |
|---|-----------------|----------------|-------------------|

select In the generated part, the derived class retrieves all the rows from a table that match the criteria specified by the clause string, an SQL **where** clause. The keyword is added by the class library code. The clause may be followed by additional parameters, as permitted in select clauses by the datastore.

| | | | |
|---|-----------------|----------------|-------------------|
| sequence < PersistentObject > select(in string clause); | <u>Win</u> Y | <u>PM</u> Y | <u>Motif</u> N |
|---|-----------------|----------------|-------------------|

Data Access Builder SOM Exceptions

The following are the SOM exceptions in Data Access Builder:

da_exception

A structure comprised of all the information that is contained in a Data Access Builder SOM exception.

ConnectFailed

Set when a database connection attempt fails.

DatastoreAccessError

Set when an error occurs during an attempt to access data in the datastore.

DatastoreConnectionInUse

Set when a connection is attempted using a connection that is already in use.

DatastoreConnectionNotOpen

Set when an operation that requires a connection is attempted before a connection has been established. An example is a call to disconnect before a connection was made.

DatastoreLogoffFailed

Set when a logoff fails.

DatastoreLogonFailed

Set when a logon attempt fails.

DataObjectAlreadyExists

Set when the data object specified in an add operation already exists in the datastore.

DataObjectAttributeError

Set when an attempt is made to set an attribute with an invalid value. An example is setting a string attribute longer than the database permits.

DataObjectInvalid

Set when an attempt to update or to delete an object results in changes to more than one row in the datastore.

DataObjectNotFound

Set when the data object specified in a retrieve, update, or delete operation cannot be found in the datastore.

DisconnectError

Set when a disconnection error occurs.

Header File

The Data Access Builder SOM exceptions are declared in `sdsexc.idl`.

da_exception Structure

POFactory

```
#define da_exception {      \
    long error_number;      \
    char error_code[8];     \
    char error_state[8];    \
    BOOLEAN error_avail;    \
    char error_text[256];   \
}
```

The following diagnostic information is passed in an exception:

error_number

Identifier for the exception.

error_code Sqlca field that contains the sqlcode.

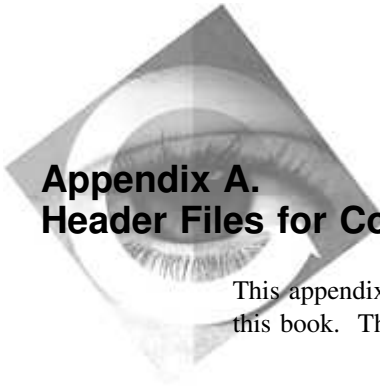
error_state Sqlca field that contains the sqlstate.

error_avail

A Boolean flag to indicate whether a request to access the database has been issued.

error_text Text describing the error.

Part 10. Appendix, Glossary, Bibliography, and Index



Appendix A. 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 | 674 | parcel.h | 681 |
| circle.h | 675 | planet.h | 682 |
| curve.h | 676 | toyword.h | 683 |
| demoelem.h | 677 | transelm.h | 683 |
| dsur.h | 678 | trmapops.h | 684 |
| line.h | 680 | xebc2asc.h | 685 |
| graph.h | 679 | | |

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;
    }

};
```


Example Header Files

```
// 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;
    }
};
```

Example Header Files

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;
    }
}
```

Example Header Files

```
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 <globals.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
```

Example Header Files

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;
    }
}
```

Example Header Files

```
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);
    }
}
```

Example Header Files

```
    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 <globals.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() << ".";
        }
    }
};
```

Example Header Files

```
        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* pname;
    float dist, mass, bright;

public:
    // Use the compiler generated default for the copy constructor

    Planet(char* aname, float adist, float amass, float abright) :
        pname(aname), dist(adist), mass(amass), bright(abright) {}

    // For any Set we need to provide element equality.
    IBoolean operator== (Planet const& aPlanet) const
    { return pname == aPlanet.pname; }

    // For a Sorted Set we need to provide element comparison.
    IBoolean operator< (Planet const& aPlanet) const
    { return dist < aPlanet.dist; }

    char* name() { return pname; }

    IBoolean isHeavy() { return (mass > 1.0); }
    IBoolean isBright() { return (bright < 0.0); }
};

// Iterator
#include <iostream.h>

class SayPlanetName : public IApplicator<Planet> {
public:
    virtual IBoolean applyTo(Planet& p)
    { cout << " " << p.name() << " "; return True; }
};
```


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) {};
};
```

Example Header Files

```
/* 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"

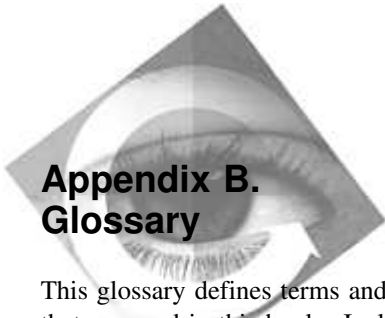
class TranslationOps : public IEOps < TranslationElement >
{
public:
    class KeyOps : public IStdEqOps < char >, public IStdHshOps < char >
    {
    } keyOps;
};

// 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
};
```



Appendix B. Glossary

This glossary defines terms and abbreviations that are used in this book. Included are terms and definitions from the following sources:

- *American National Standard Dictionary for Information Systems*, American National Standard for Information Systems X3.172-1990, copyright 1990 by the American National Standards Institute (American National Standard for Information Systems). Copies may be purchased from the American National Standards Institute, 1430 Broadway, New York, New York 10018. Such definitions are indicated by the symbol *American National Standard for Information Systems* after the definition.
- *IBM Dictionary of Computing*, SC20-1699. These definitions are indicated by the registered trademark *IBM* after the definition.
- *X/Open CAE Specification. Commands and Utilities, Issue 4. July, 1992*. These definitions are indicated by the symbol *X/Open* after the definition.
- *ISO/IEC 9945-1:1990/IEEE POSIX 1003.1-1990*. These definitions are indicated by the symbol *ISO.1* after the definition.
- *The Information Technology Vocabulary*, developed by Subcommittee 1, Joint Technical Committee 1, of the International Organization for Standardization and the International Electrotechnical Commission (ISO/IEC JTC1/SC1). Definitions of published parts of this vocabulary are identified by the symbol *ISO-JTC1* after the definition; definitions taken from draft international standards, committee drafts, and working papers being developed by ISO/IEC JTC1/SC1 are identified by the symbol *ISO Draft* after the definition, indicating that final agreement has not yet been reached among the participating National Bodies of SC1.

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 •array element

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.

action. A description of tool or function that can be used to manipulate a project's parts, or build a project's target. Examples are compile, link, and edit.

address space. (1) The range of addresses available to a computer program. *American National Standard for Information Systems.* (2) The complete range of addresses that are available to a programmer. (3) The area of virtual storage available for a particular job. (4) The memory locations that can be referenced by a process. *X/Open. ISO.1.*

aggregate. (1) An array or a structure. (2) A compile-time option to show the layout of a structure or union in the listing. (3) An array or a class object with no private or protected members, no constructors, no base classes, and no virtual functions. (4) In programming languages, a structured collection of data items that form a data type. *ISO-JTC1.*

alignment. The storing of data in relation to certain machine-dependent boundaries. *IBM.*

ambiguous derivation. A derivation where the class is derived from two or more base classes that have members with the same name.

American National Standards Institute. See *American National Standard for Information Systems.*

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.

American National Standard for Information Systems (American National Standards Institute). An organization consisting of producers, consumers, and general interest groups, that establishes the procedures by which accredited organizations create and maintain voluntary industry standards in the United States. *American National Standard for Information Systems.*

application. (1) The use to which an information processing system is put; for example, a payroll application, an airline reservation application, a network application. *IBM.* (2) A collection of software components used to perform specific types of user-oriented work on a computer. *IBM.*

array. An aggregate that consists of data objects, with identical attributes, each of which may be uniquely referenced by subscripting.

array element. A data element in an array.

array implementation •block

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 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 placed 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.

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.

block. (1) In programming languages, a compound statement that coincides with the

scope of at least one of the declarations contained within it. A block may also specify storage allocation or segment programs for other purposes. *ISO-JTC1*. (2) A string of data elements recorded or transmitted as a unit. The elements may be characters, words, or physical records. *ISO Draft*. (3) The unit of data transmitted to and from a device. Each block contains one record, part of a record, or several records.

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.

brackets. The characters [(left bracket) and] (right bracket), also known as *square brackets*. When used in the phrase “enclosed in (square) brackets” the symbol [immediately precedes the object to be enclosed, and] immediately follows it. When describing these characters in the portable character set, the names <left-bracket> and <right-bracket> are used. *X/Open*.

build. An action that invokes the WorkFrame Build tool. The Build tool manages the project's makefile, as well as build dependencies between projects in a project hierarchy.

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. Synonymous with predefined. *IBM*.

C

C++ class library. See *class library*.

C++ library. A system library that contains common C++ language subroutines for file access, memory allocation, and other functions.

call. To transfer control to a procedure, program, routine, or subroutine. *IBM*.

caller. A routine that calls another routine.

camcorder. A compact, hand-held video camera with integrated videotape recorder.

canvas. Canvases are windows with a layout algorithm that manages child windows. The canvas classes are a set of window classes that allow you to implement dialog boxes. These dialog 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*.

CASE. Computer-Aided Software Engineering.

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-XA. Compact disc-extended architecture.

channel mapping • C library

channel mapping. The translation of a MIDI channel number for a sending device to an appropriate channel for a receiving device.

character. (1) A letter, digit, or other symbol that is used as part of the organization, control, or representation of data. A character is often in the form of a spatial arrangement of adjacent or connected strokes. *American National Standard for Information Systems.* (2) A sequence of one or more bytes representing a single graphic symbol or control code. This term corresponds to the ISO C standard term *multibyte character* (multi-byte character), where a single-byte character is a special case of the multi-byte character. Unlike the usage in the ISO C standard, *character* here has no necessary relationship with storage space, and *byte* is used when storage space is discussed. *X/Open. ISO.1.*

character array. An array of type `char`. *X/Open.*

character set. (1) A finite set of different characters that is complete for a given purpose; for example, the character set in ISO Standard 646, 7-bit Coded Character Set for Information Processing Interchange. *ISO Draft.* (2) All the valid characters for a programming language or for a computer system. *IBM.* (3) A group of characters used for a specific reason; for example, the set of characters a printer can print. *IBM.* (4) See also *portable character set.*

character string. A contiguous sequence of characters terminated by and including the first null byte. *X/Open.*

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. (1) A group of objects that share a common definition and that therefore share common properties, operations, and behavior. (2) A C++ aggregate that may contain functions, types, and user-defined operators in addition to data. 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 key. Any of the three C++ keywords: **class**, **struct**, or **union**.

class library. A collection of classes.

class template. A blueprint describing how a set of related classes can be constructed.

class name. A unique identifier of a class type that becomes a reserved word within its scope.

client area window. An intermediate window between an `IFrameWindow` and its controls and other child windows.

C library. A system library that contains common C language subroutines for file access, string operators, character operations, memory allocation, and other functions. *IBM.*

client program •Compound Object Model (COM)

client program. A program that uses a class. The program is said to be a client of the class.

coded character set. (1) A set of graphic characters and their code point assignments. The set may contain fewer characters than the total number of possible characters: some code points may be unassigned. *IBM.* (2) A coded set whose elements are single characters; for example, all characters of an alphabet. *ISO Draft.* (3) Loosely, a code. *American National Standard for Information Systems.*

code page. (1) An assignment of graphic characters and control function meanings to all code points; for example, assignment of characters and meanings to 256 code points for an 8-bit code, or assignment of characters and meanings to 128 code points for a 7-bit code. (2) A particular assignment of hexadecimal identifiers to graphic characters.

collating sequence. (1) A specified arrangement used in sequencing. *ISO-JTC1. American National Standard for Information Systems.* (2) An ordering assigned to a set of items, such that any two sets in that assigned order can be collated. *American National Standard for Information Systems.* (3) The relative ordering of collating elements as determined by the setting of the LC_COLLATE category in the current locale. The character order, as defined for the LC_COLLATE category in the current locale, defines the relative order of all collating elements, such that each element occupies a unique position in the order. This is the order used in ranges of characters and collating elements in regular expressions and pattern matching. In addition, the definition of the collating weights of characters and collating elements uses collating elements to represent their respective positions within the collation sequence.

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 Class Library. A set of classes that provide basic functions for collections, and can be used as base classes.

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.

COM. Component Object Model.

command. A request to perform an operation or run a program. When parameters, arguments, flags, or other operands are associated with a command, the resulting character string is a single command.

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). 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.

Compound Object Model (COM). The underlying model for all OLE services. It consists of a variety of APIs and object interfaces

composite •conversion

that allow container components to communicate and interact with one another.

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.

condition. (1) A relational expression that can be evaluated to a value of either true or false. *IBM.* (2) An exception that has been enabled, or recognized, by the *Language Environment* and thus is eligible to activate user and language condition handlers. Any alteration to the normal programmed flow of an application. Conditions can be detected by the hardware/operating system and result in an interrupt. They can also be detected by language-specific generated code or language library code.

conditional expression. A compound expression that contains a condition (the first expression), an expression to be evaluated if the condition has a nonzero value (the second expression), and an expression to be evaluated if the condition has the value zero (the third expression).

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.

constant. (1) In programming languages, a language object that takes only one specific

value. *ISO-JTC1.* (2) A data item with a value that does not change. *IBM.*

constant expression. An expression having a value that can be determined during compilation and that does not change during the running of the program. *IBM.*

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.

container. An object that holds other objects.*IBM.* Containers built with the Compound Document Framework are also servers and can therefore be embedded inside other containers. A container can hold zero or more embedded components.

containment function. A function that determines whether a collection contains a given element.

control. A graphic object that represents operations or properties of other objects. See also tree control.

control character. (1) A character whose occurrence in a particular context specifies a control function. *ISO Draft.* (2) Synonymous with nonprinting character. *IBM.* (3) A character, other than a graphic character, that affects the recording, processing, transmission, or interpretation of text. *X/Open.*

conversion. (1) In programming languages, the transformation between values that represent the same data item but belong to different data types. Information may be lost because of conversion since accuracy of data representation varies among different data types. *ISO-JTC1.* (2) The process of changing from one method of data processing to another or from one data

conversion function •DBCS (Double-Byte Character Set)

processing system to another. *IBM*. (3) The process of changing from one form of representation to another; for example to change from decimal representation to binary representation. *IBM*. (4) A change in the type of a value. For example, when you add values having different data types, the compiler converts both values to a common form before adding the values.

conversion function. A member function that specifies a conversion from its class type to another type.

copy constructor. A constructor used to make a copy of an object from another object of the same type.

critical section. (1) Code that must be executed by one thread while all other threads in the process are suspended. (2) In Windows, a synchronization object. A critical section is not a kernel object; that is, it is not managed by the low-level components of the operating system and is not manipulated using handles. (3) In Windows, a small section of code that requires exclusive access to some shared data before the code can execute. Critical threads synchronize threads only within a single process, and they allow only one thread at a time to gain access to a region of data.

See also mutex, semaphore, and event. Contrast with kernel object.

C/2. A version of the C language designed for the OS/2 environment.

current working directory. (1) A directory, associated with a process, that is used in path-name resolution for path names that do not begin with a slash. *X/Open. ISO.1*. (2) In DOS, the directory that is searched when a file name is entered with no indication of the directory that

lists the file name. DOS assumes that the current directory is the root directory unless a path to another directory is specified. *IBM*. (3) In the OS/2 operating system, the first directory in which the operating system looks for programs and files and stores temporary files and output. *IBM*. (4) In the AIX operating system, a directory that is active and that can be displayed. Relative path name resolution begins in the current directory. *IBM*.

cursor. A reference to an element at a specific position in a data structure.

cursor emphasis. When the selection cursor is on a choice, that choice has cursor emphasis.

cursor iteration. The process of repeatedly moving the cursor to the next element in a collection until some condition is satisfied.

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.

data member. The smallest possible piece of complete data. Elements are composed of data members.

data structure. The internal data representation of an implementation.

data type. The properties and internal representation that characterize data.

DBCS (Double-Byte Character Set). See double-byte character set.

decimal constant •device

decimal constant. (1) A numerical data type used in standard arithmetic operations. (2) A number containing any of the digits 0 through 9. *IBM.*

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 interpretation 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.

define directive. A preprocessor statement that directs the preprocessor to replace an identifier or macro invocation with special code.

definition. (1) A data description that reserves storage and may provide an initial value. (2) A declaration that allocates storage, and may initialize a data object or specify the body of a function.

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.

device. A computer peripheral or an object that appears to the application as such. *X/Open. ISO.1.*

difference • dump

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. (1) 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. (2) In Windows, a unit of data. The unit is often part of a task that is shared among users.

directory. A type of file containing the names and controlling information for other files or other directories. *IBM.*

display. To direct the output to the user's terminal. If the output is not directed to the terminal, the results are undefined. *X/Open.*

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.

dump. To copy data in a readable format from main or auxiliary storage onto an external medium such as tape, diskette, or printer. *IBM.*

dynamic •event

dynamic. Pertaining to an operation that occurs at the time it is needed rather than at a predetermined or fixed time. *IBM.*

dynamic binding. Resolution of a call to a virtual member function at run time.

dynamic link library (DLL). A file containing executable code and data bound to a program at load time or run time. The code and data in a dynamic link library can be shared by several applications simultaneously.

dynamic storage. Synonym for *automatic storage*.

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.

empty string. (1) A string whose first byte is a null byte. Synonymous with null string. *X/Open.* (2) A character array whose first element is a null character. *ISO.I.*

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.

enumerator. In the C and C++ language, an enumeration constant and its associated value. *IBM.*

environment variable. Any of a number of variables that describe the way an operating system is going to run or the devices it is going to recognize. *IBM.*

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.

event. (1) Any user action (such as a mouse click) or system activity (such as screen updating) that provokes a response from the application. (2) In Windows, a synchronization kernel object used to signal that an operation has

completed. See also kernel object. Compare to critical section, mutex, semaphore, manual-reset event, and auto-reset event.

exception. (1) A user or system error detected by the system and passed to an operating system 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.

extension. (1) An element or function not included in the standard language. (2) File name extension.

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. (1) 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. (2) In WorkFrame, the value of a type. The filter of a type can be expressed as a file mask; a regular expression; a logical-OR, logical-AND, or logical-NOT of a list of types; or a filter determined by a PAM.

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.

folder. A directory.

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 •graphic primitive

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.

free store. Dynamically allocated memory. New is used to allocate free store and delete to deallocate it.

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.

function. A named group of statements that can be called and evaluated and can return a value to the calling statement. *IBM.*

function call. An expression that moves the path of execution from the current function to a

specified function and evaluates to the return value provided by the called function. A function call contains the name of the function to which control moves and a parenthesized list of values. *IBM.*

function definition. The complete description of a function. A function definition contains an optional storage class specifier, an optional type specifier, a function declarator, optional parameter declarations, and a block statement (the function body).

function prototype. A function declaration that provides type information for each parameter. It is the first line of the function (header) followed by a ; (semicolon). The declaration is required by the compiler at the time that the function is declared, so that the compiler can check the type.

G

gain. The ability to change the audibility of the sound, such as during a fade in or fade out of music.

global. Pertaining to information available to more than one program or subroutine. *IBM.*

global scope. See *file scope*.

global variable. A symbol defined in one program module that is used in other independently compiled program modules.

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) • inheritance

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.

identifier. (1) One or more characters used to identify or name a data element and possibly to indicate certain properties of that data element. *American National Standard for Information Systems.* (2) In programming languages, a token that names a data object such as a variable, an array, a record, a subprogram, or a function. *American National Standard for Information Systems.* (3) A sequence of letters, digits, and underscores used to identify a data object or function. *IBM.*

implementation class. A class that implements a concrete class. Implementation classes are never used directly.

include file. See *header file*.

include statement. In the C and C++ languages, a preprocessor statement that causes the preprocessor to replace the statement with the contents of a specified file. *IBM.*

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. (1) A mechanism for connecting objects by storing, in one object, a reference to another object. (2) In the C and C++ languages, the application of the unary operator * to a pointer to access the object the pointer points to.

inheritance. (1) An object-oriented programming technique that allows you to use existing classes as bases for creating other classes. (2) 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.

initializer • I/O Stream Library

initializer. An expression used to initialize objects. In the C++ language, there are three types of initializers:

1. An expression followed by an assignment operator is used to initialize fundamental data type objects or class objects that have copy constructors.
2. An expression enclosed in braces ({ }) is used to initialize aggregates.
3. A parenthesized expression list is used to initialize base classes and members using constructors.

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 sequence of control statements and data submitted to a system from an input unit.

instance (of a class). An object that is a member of that class. An object created according to the definition of that class.

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.

instantiate. To create or generate a particular instance or object of a data type. For example, an instance box1 of class box could be instantiated with the declaration:

```
box box1;
```

instruction. A program statement that specifies an operation to be performed by the computer,

along with the values or locations of operands. This statement represents the programmer's request to the processor to perform a specific operation.

integer constant. A decimal, octal, or hexadecimal constant.

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.

internationalization. The capability of a computer program to adapt to the requirements of different native languages, local customs, and coded character sets. *X/Open*.

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.

I/O Stream Library. A class library that provides the facilities to deal with many varieties of input and output.

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.

K

kernel. The core of an operating system, usually responsible for basic I/O and process execution.

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

label. An identifier within or attached to a set of data elements.

ISO Draft.

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.

late binding. See *dynamic binding*.

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

link •mask

object modules that can be specified in a link command.

link. To interconnect items of data or portions of one or more computer programs; for example, linking of object programs by a linkage editor to produce an executable file.

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.

literal. (1) In programming languages, a lexical unit that directly represents a value; for example, 14 represents the integer fourteen, “APRIL” represents the string of characters APRIL, 3.0005E2 represents the number 300.05. *ISO-JTC1.* (2) A symbol or a quantity in a source program that is itself data, rather than a reference to data. *IBM.* (3) A character string whose value is given by the characters themselves; for example, the numeric literal 7 has the value 7, and the character literal CHARACTERS has the value CHARACTERS. *IBM.*

loader. A routine, commonly a computer program, that reads data into main storage. *American National Standard for Information Systems.*

load module. All or part of a computer program in a form suitable for loading into main storage for execution. A load module is usually the output of a linkage editor. *ISO Draft.*

local. (1) In programming languages, pertaining to the relationship between a language object and a block such that the language object has a scope contained in that block. *ISO-JTC1.*
(2) Pertaining to that which is defined and used only in one subdivision of a computer program. *American National Standard for Information Systems.*

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

macro. An identifier followed by arguments (may be a parenthesized list of arguments) that the preprocessor replaces with the replacement code located in a preprocessor #define directive.

main function. An external function with the identifier main that is the first user function—aside from exit routines and C++ static object constructors—to get control when program execution begins. Each C and C++ program must have exactly one function named main.

make. An action in which a project's target is built from a makefile by a make utility.

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

MBCS •multibyte character set (MBCS)

portions of another pattern of bits or characters.
ISO-JTC1. ANSI.

MBCS. See multibyte character set.

member. Data, functions, or types contained in classes, structures, or unions.

member function. (1) 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. Member functions are also called methods. (2) A function that performs operations on a 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.

migrate. To move to a changed operating environment, usually to a new release or version of a system. *IBM.*

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. *X/Open. ISO.1.*

model. In Compound Document Framework, the data portion of a document component. The model and the view comprise the two pieces of a document component. Compound Document Framework provides an IModel base class from which other model classes can be derived.

module. A program unit that usually performs a particular function or related functions, and that is distinct and identifiable with respect to compiling, combining with other units, and loading.

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. A mixture of single-byte characters from a single-byte character set and double-byte characters from a double-byte character set.

multibyte character set (MBCS). A character set whose characters consist of more than 1 byte. Used in languages such as Japanese, Chinese,

multimedia • NTSC format

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. (1) A mode of operation that allows concurrent performance or interleaved execution of more than one task or program. (2) A process that allows a computer or operating system to run multiple applications or tasks concurrently by dividing the processor's time between them rapidly.

See also preemptive multitasking. Contrast with nonpreemptive multitasking.

multithread. Pertaining to concurrent operation of more than one path of execution within a computer.

N

name. In the C++ language, a name is commonly referred to as an identifier. However, syntactically, a name can be an identifier, operator function name, conversion function name, destructor name, or qualified name.

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

NULL •ordered collection

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. In the C and C++ languages, a pointer that does not point to a data object. *IBM.*

null character (\0). The ASCII or EBCDIC character with the hex value 00 (all bits turned off). It is used to represent the absence of a printed or displayed character. This character is named <NUL> in the portable character set.

null pointer. The value that is obtained by converting the number 0 into a pointer; for example, (void *) 0. The C and C++ languages guarantee that this value will not match that of any legitimate pointer, so it is used by many functions that return pointers to indicate an error. *X/Open.*

null string. (1) A string whose first byte is a null byte. Synonymous with *empty string*. *X/Open.* (2) A character array whose first element is a null character. *ISO.1.*

O

object. (1) A computer representation of something that a user can work with to perform a task. An object can appear as text or an icon. (2) 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 member functions. Each object in the class is said to be an instance of the class. (3) In Visual Builder, an instance of an object class consisting of attributes, a data structure, and operational member functions. It can represent a

person, place, thing, event, or concept. Each instance has the same properties, attributes, and member functions as other instances of the object class, though it has unique values assigned to its attributes. (4) In Windows, any item that is or can be linked into another Windows application, such as a sound, graphic, piece of text, or portion of a spreadsheet. An object must be from an application that supports OLE. See object linking and embedding (OLE).

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.

open file. A file that is currently associated with a file descriptor. *X/Open. ISO.1.*

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.

operator precedence. In programming languages, an order relation defining the sequence of the application of operators within an expression. *ISO-JTC1.*

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

ordering relation •pause

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

pack. To store data in a compact form in such a way that the original form can be recovered.

pad. To fill unused positions in a field with data, usually 0's, 1's, or blanks.

parameter. (1) In the C and C++ languages, an object declared as part of a function declaration or definition that acquires a value on entry to the function, or an identifier following the macro name in a function-like macro definition. *X/Open*. (2) Data passed between programs or procedures. *IBM*.

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.

part. In Visual Builder, a part is a self-contained object with a standardized public interface consisting of a set of external features that allow the part to interact with other parts. A part is implemented as a class that supports the INotifier protocol and has a part interface defined.

path name. (1) A string that is used to identify a file. A path name consists of, at most, {PATH_MAX} bytes, including the terminating null character. It has an optional beginning slash, followed by zero or more file names separated by slashes. If the path name refers to a directory, it may also have one or more trailing slashes. Multiple successive slashes are considered to be the same as one slash. A path name that begins with two successive slashes may be interpreted in an implementation-dependent manner, although more than two leading slashes will be treated as a single slash. The interpretation of the path name is described in *pathname resolution. ISO.1*. (2) A file name specifying all directories leading to the file.

pattern. A sequence of characters used either with regular expression notation or for path name expansion, as a means of selecting various characters strings or path names, respectively. The syntaxes of the two patterns are similar, but not identical. *X/Open*.

pause. To temporarily halt the medium. The halted visual should remain displayed but no audio should be played.

pel •portable character set

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.

pipe. To direct data so that the output from one process becomes the input to another process. The standard output of one command can be connected to the standard input of another with the pipe operator (`|`). Two commands connected in this way constitute a pipeline. *IBM.*

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.

portable character set. The set of characters specified in POSIX 1003.2, section 2.4:

| | |
|---------------------|----|
| <NUL> | |
| <alert> | |
| <backspace> | |
| <tab> | |
| <newline> | |
| <vertical-tab> | |
| <form-feed> | |
| <carriage-return> | |
| <space> | |
| <exclamation-mark> | ! |
| <quotation-mark> | " |
| <number-sign> | # |
| <dollar-sign> | \$ |
| <percent-sign> | % |
| <ampersand> | & |
| <apostrophe> | ' |
| <left-parenthesis> | (|
| <right-parenthesis> |) |
| <asterisk> | * |
| <plus-sign> | + |
| <comma> | , |
| <hyphen> | - |
| <hyphen-minus> | - |
| <period> | . |
| <slash> | / |
| <zero> | 0 |
| <one> | 1 |
| <two> | 2 |
| <three> | 3 |
| <four> | 4 |
| <five> | 5 |
| <six> | 6 |
| <seven> | 7 |
| <eight> | 8 |
| <nine> | 9 |
| <colon> | : |
| <semicolon> | ; |
| <less-than-sign> | < |
| <equals-sign> | = |
| <greater-than-sign> | > |
| <question-mark> | ? |
| <commercial-at> | @ |

portability • predicate function

| | | | |
|------------------------|---|-----------------------|---|
| <A> | A | <a> | a |
| | B | | b |
| <C> | C | <c> | c |
| <D> | D | <d> | d |
| <E> | E | <e> | e |
| <F> | F | <f> | f |
| <G> | G | <g> | g |
| <H> | H | <h> | h |
| <I> | I | <i> | i |
| <J> | J | <j> | j |
| <K> | K | <k> | k |
| <L> | L | <l> | l |
| <M> | M | <m> | m |
| <N> | N | <n> | n |
| <O> | O | <o> | o |
| <P> | P | <p> | p |
| <Q> | Q | <q> | q |
| <R> | R | <r> | r |
| <S> | S | <s> | s |
| <T> | T | <t> | t |
| <U> | U | <u> | u |
| <V> | V | <v> | v |
| <W> | W | <w> | w |
| <X> | X | <x> | x |
| <Y> | Y | <y> | y |
| <Z> | Z | <z> | z |
| <left-square-bracket> | [| <left-brace> | { |
| <backslash> | \ | <left-curly-bracket> | { |
| <reverse-solidus> | \ | <vertical-line> | |
| <right-square-bracket> |] | <right-brace> | } |
| <circumflex> | ^ | <right-curly-bracket> | } |
| <circumflex-accent> | ^ | <tilde> | ~ |
| <underscore> | _ | | |
| <low-line> | ~ | | |
| <grave-accent> | ` | | |

portability. The ability of a programming language to compile successfully on different operating systems without requiring changes to the source code.

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.

precedence. The priority system for grouping different types of operators with their operands.

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 •public

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. (1) A collection of code, data, and other system resources, including at least one thread of execution, that performs a data processing task. (2) A running application, its address space, and its resources. (3) An instance of a running program. A Win32 process owns a 4-GB address space containing the code and data for an application's .exe file; it does not execute anything. It also owns certain resources, such as files, dynamic memory allocations, and threads. (4) 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.

project. (1) A container that groups related objects (tasks) into a primary window. When the user opens the object, the object has its own primary window. (2) In WorkFrame, the complete set of data objects (called project parts) and actions needed to build a single target, such as a dynamic link library (DLL) or executable file (EXE).

Project Smarts. A project catalog that contains templates for common types of applications.

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 •samples-per-second

pure virtual function. A virtual function that has a function initializer of the form `= 0;`.

Q

qualified name. Used to qualify a nonclass type name such as a member by its class name.

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

redirection. In the shell, a method of associating files with the input or output of commands. *X/Open*.

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.

return. A language construct that ends an execution sequence in a procedure. *IBM*.

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.

RTTI. Run-time type identification.

run-time type identification (RTTI). A mechanism in the C++ language for determining the class of an object at run time. It consists of two operators, one for determining the run-time type of an object (**typeid**) and one for doing type conversions that are checked at run time (**dynamic_cast**). A **type_info** class describes the RTTI available and defines the type returned by the **typeid** operator.

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 (Single-Byte Character Set) •sorted map

SBCS (Single-Byte Character Set). See single-byte character set.

scalar. An arithmetic object, or a pointer to an object of any type.

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.

server. In Compound Document Framework, an application or a document component that supplies an object. For example, a drawing program that provides a picture that can be placed inside a word processing document is referred to as a server.

shell. A program that interprets sequences of text input as commands. It may operate on an input stream or it may interactively prompt and read commands from a terminal. *X/Open*.

This feature is provided as part of OpenEdition MVS Shell and Utilities feature licensed program.

siblings. All the children of a node are said to be siblings of one another.

signal. (1) A condition that may or may not be reported during program execution. For example, SIGFPE is the signal used to represent erroneous arithmetic operations such as a division by zero. (2) A mechanism by which a process may be notified of, or affected by, an event occurring in the system. Examples of such events include hardware exceptions and specific actions by processes. The term *signal* is also used to refer to the event itself. *X/Open*. *ISO.1*. (3) In AIX operating system operations, a method of interprocess communication that simulates software interrupts. *IBM*.

single-byte character set (SBCS). A set of characters in which each character is represented by a 1-byte code.

slash. The character /, also known as *solidus*. This character is named <slash> in the portable character set.

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 •stream

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.

source file. A file that contains source statements for such items as high-level language programs and data description specifications. *IBM.*

source program. A set of instructions written in a programming language that must be translated to machine language before the program can be run. *IBM.*

space character. The character defined in the portable character set as <space>. The space character is a member of the space character class of the current locale, but represents the single character, and not all of the possible members of the class. *X/Open.*

specifiers. Used in declarations to indicate storage class, fundamental data type and other properties of the object or function being declared.

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.

stack storage. Synonym for *automatic storage*.

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. *X/Open.* 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.

statement. An instruction that ends with the character ; (semicolon) or several instructions that are surrounded by the characters { and }.

static. A keyword used for defining the scope and linkage of variables and functions. For internal variables, the variable has block scope and retains its value between function calls. For external values, the variable has file scope and retains its value within the source file. For class variables, the variable is shared by all objects of the class and retains its value within the entire program.

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.

string literal. Zero or more characters enclosed in double quotation marks.

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.

subsystem. A secondary or subordinate system, usually capable of operating independently of or asynchronously with, a controlling system. *ISO Draft.*

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.

target. A WorkFrame project's target is the file that is produced as a result of a project build.

task. (1) In a multiprogramming or multiprocessing environment, one or more sequences of instructions treated by a control program as an element of work to be accomplished by a computer. *ISO-JTC1. American National Standard for Information Systems.* (2) A routine that is used to simulate the operation of programs. Tasks are said to be *nonpreemptive* because only a single task is executing at any one time. Tasks are said to be *lightweight* because less time and space are required to create a task than a true operating system process.

task library. A class library that provides the facilities to write programs that are made up of tasks.

template. A family of classes or functions where the code remains invariant but operates with variable types.

terminals. Synonym for *leaves*.

template class. A class instance generated by a class template.

template function. A function generated by a function template.

text file •type

text file. A file that contains characters organized into one or more lines. The lines must not contain null characters and none can exceed {LINE_MAX}—which is defined in limits.h—bytes in length, including the new-line character. The term *text file* does not prevent the inclusion of control or other nonprintable characters (other than NUL). *X/Open*.

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. (1) The smallest unit or path of execution within a process. *IBM*. (2) A piece of executing code. (3) In Windows, each thread is allocated its own stack from the owning process' 4-GB address space, and each one has its own set of processor registers, called the thread's context. See also primary thread and zero page thread.

throw expression. An argument to the C++ exception being thrown.

tilde. The character ~. This character is named <tilde> in the portable character set.

time code. See SMPTE time code.

tool bar. The area under the title bar that displays the tools available.

token. The smallest independent unit of meaning of a program as defined either by a parser or a lexical analyzer. A token can contain data, a language keyword, an identifier, or other parts of language syntax.

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. *ISO-JTC1*.

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. (1) A block in which a known C++ exception is passed to a handler. (2)

type. (1) The description of the data and the operations that can be performed on or by the data. See also *data type*. (2) In WorkFrame, describes a group of project files of parts in terms of an expression, such as file masks,

type class •unrecoverable error

regular expressions, or a list of other types, logical-OR'd.

type class. In WorkFrame, represents the method by which an object is determined to be a member of a type. "File mask" is an example of a type class. Membership to a "file mask" type is determined by matching the file mask filter to the object's name. Other examples of type classes are "regular expression," and "PAM name," where the named Project Access Method determines membership to a type.

type definition. A definition of a name for a data type. *IBM.*

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.

type specifier. Used to indicate the data type of an object or function being declared.

U

undefined behavior. Referring to a program or function that may produce erroneous results without warning because of its use of an indeterminate value, or because of erroneous program constructs or erroneous data.

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 bytes in memory.

unbounded collection. A collection that has no upper limit on the number of elements it can contain.

undefined behavior. Referring to a program or function that may produce erroneous results without warning because of its use of an indeterminate value, or because of erroneous program constructs or erroneous data.

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 it 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. *IBM.*

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. *IBM.* (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.

unrecoverable error. An error for which recovery is impossible without use of recovery

variable • Windows NT

techniques external to the computer program or run.

V

variable. In programming languages, a language object that may take different values, one at a time. The values of a variable are usually restricted to a certain data type. *ISO-JTC1*.

VCR. Videocassette recorder.

VGA. Video graphics adapter.

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 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.

view. In Compound Document Framework, the user interface controls and associated handlers for a document component. The view and the model comprise the two pieces of a document component. The Compound Document

Framework provides an IView base class from which other view classes can be derived.

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.

visible. Visibility of identifiers is based on scoping rules and is independent of *access*.

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.

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.

Win32. The name of a 32-bit application programming interface (API) developed by Microsoft.

See also Win32 API.

Windows NT. An operating system that the Win32 API is implemented on. It is a portable, high-end operating system, which supports multitasking. It is the only operating system that allows implementation of the Win32 APIs on machine architectures based on processors other than the x86, and it supports multiple processors.

See also Win32 API.

word boundary •(::) (double colon)

word boundary. Any storage position at which data must be aligned for certain processing operations. The halfword boundary must be divisible by 2; the fullword boundary by 4; and the doubleword boundary by 8. *IBM.*

working directory. (1) Synonym for *current working directory*. (2) The directory where files that are copied or dragged into the project are stored. Actions are also executed in this directory, so this directory is where many output files are placed.

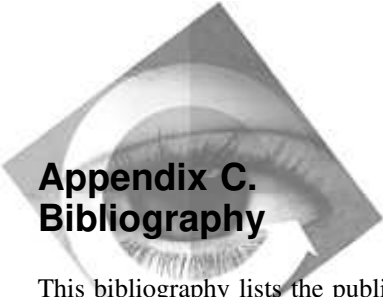
write. (1) To output characters to a file, such as standard output or standard error. Unless otherwise stated, standard output is the default output destination for all uses of the term *write*. *X/Open*. (2) To make a permanent or transient recording of data in a storage device or on a data

medium. *ISO-JTC1. American National Standard for Information Systems.*

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).

(::) (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.



Appendix C. Bibliography

This bibliography lists the publications that make up the IBM VisualAge for C++ library and related publications. The list of related publications is not exhaustive but should be adequate for most VisualAge for C++ users.

The IBM VisualAge for C++ Library

The following books are part of the IBM VisualAge for C++ library.

- *Installation Guide and Product Overview*, S33H-5030
- *User's Guide*, S33H-5031
- *Programming Guide*, S33H-5032
- *Visual Builder User's Guide*, S33H-5034
- *Visual Builder Parts Reference*, S33H-5035
- *Building VisualAge for C++ Parts for Fun and Profit*, S33H-5036
- *Open Class Library User's Guide*, S33H-5033
- *Open Class Library Reference*, S33H-5039
- *Language Reference*, S33H-5037
- *C Library Reference*, S33H-5038
- *SOM Programming Guide*, S33H-5043
- *SOM Programming Reference*, S33H-5044

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])*

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.

Appendix D. Index

Special Characters

- ~Cursor
 - IObserverList::Cursor 450
- ~IOString
 - IOString 290
- ~IBaseErrorInfo
 - IBaseErrorInfo 317
- ~IBaseStream
 - IBaseStream 320
- ~IBuffer
 - IBuffer 355
- ~ICLibErrorInfo
 - ICLibErrorInfo 359
- ~IContext
 - IContext 363
- ~IDatastore
 - IDatastore 611
- ~IDatastoreBase
 - IDatastoreBase 617
- ~IDatastoreDB2
 - IDatastoreDB2 625
- ~IDatastoreODBC
 - IDatastoreODBC 633
- ~IDBCSBuffer
 - IDBCSBuffer 388
- ~IException
 - IException 397
- ~IFileStream
 - IFileStream 409
- ~IGUIErrorInfo
 - IGUIErrorInfo 415
- ~IMemoryStream
 - IMemoryStream 423
- ~IMessageText
 - IMessageText 428
- ~IMetaType
 - IMetaType 430
- ~INotificationEvent
 - INotificationEvent 436
- ~INotifier
 - INotifier 440
- ~IObserver
 - IObserver 444
- ~IObserverList
 - IObserverList 447
- ~IPersistentObject
 - IPersistentObject 640
- ~IPointArray
 - IPointArray 471
- ~IPOManager
 - IPOManager 645
- ~IRefCounted
 - IRefCounted 493
- ~IReference
 - IReference 497
- ~IStandardNotifier
 - IStandardNotifier 504
- ~IString
 - IString 515
- ~IStringParser
 - IStringParser 564
- ~IStringTest
 - IStringTest 574
- ~ISystemErrorInfo
 - ISystemErrorInfo 581
- ~ITrace
 - ITrace 593



- TVBase
 - IVBase 599
- IXLibErrorInfo
 - IXLibErrorInfo 602
- _get_access_mode
 - DatastoreDB2 656
 - DatastoreODBC 663
- _get_authentication
 - DatastoreBase 653
- _get_auto_commit
 - DatastoreDB2 656
 - DatastoreODBC 663
- _get_datastore_name
 - DatastoreBase 653
- _get_isolation_level
 - DatastoreDB2 657
 - DatastoreODBC 663
- _get_user_name
 - DatastoreBase 653
- _set_access_mode
 - DatastoreDB2 657
 - DatastoreODBC 664
- _set_authentication
 - DatastoreBase 653
- _set_auto_commit
 - DatastoreDB2 658
 - DatastoreODBC 665
- _set_datastore_name
 - DatastoreBase 653
- _set_isolation_level
 - DatastoreDB2 658
 - DatastoreODBC 665
- _set_user_name
 - DatastoreBase 653

Numerics

- I0String 287

A

- abs() complex function 14
- abstract Collection Classes
 - collection 272
 - equality collection 273
 - equality key collection 274
 - equality key sorted collection 275
 - equality sorted collection 276
 - key collection 277
 - key sorted collection 278
 - ordered collection 279
 - sequential collection 280
 - sorted collection 281
- IAccessError 304
- accessMode
 - IDatastoreDB2 626
 - IDatastoreODBC 634
- accessModeId
 - IDatastoreDB2 630
 - IDatastoreODBC 638
- add
 - IContext 363
 - IObserverList 448
 - IPersistentObject 641
 - IPointArray 471
 - PersistentObject 668
- add() Collection Class function 101
- addAllFrom() Collection Class function 102
- addAsChild() tree function 245
- addAsFirst() Collection Class function 103
- addAsLast() Collection Class function 103
- addAsNext() Collection Class function 104
- addAsPrevious() Collection Class function 104
- addAsRoot() tree function 246
- addAtPosition() Collection Class function 105
- addDifference() Collection Class function 105
- addIntersection() Collection Class function 106
- addLocation
 - IException 398

- addObserver
 - INotifier 441
 - IStandardNotifier 506
- addOrReplaceElementWithKey() Collection Class
 - function 107
- addRef
 - IBuffer 348
 - IRefCounted 492
- addUnion() Collection Class function 107
- adjustArg
 - IOString 302
- adjustResult
 - IOString 302
- allElementsDo() function
 - flat collections 108—109
 - tree collections 246—247
- allocate
 - IBuffer 354
 - IDBCSBuffer 377
- allocate() function
 - streambuf class 81
- allSubtreeElementsDo() tree function 246—247
- anyElement() Collection Class function 109
- appendText
 - IException 400
- applicator class 265
- applyBitOp
 - IOString 547
- area
 - IRectangle 481
- arg() complex function 14
- arg1, arg2 c_exception arguments 16
- array initialization in complex class 10
- array stream buffer classes 87—90
- asDebugInfo
 - IBase 309
 - IBuffer 341
 - IPair 461
 - IRectangle 480
 - IOString 515
- asDebugInfo (*continued*)
 - IVBase 599
- asDouble
 - IOString 543
- asExtendedUnsignedLong
 - IBitFlag 338
- asICnrDate
 - IDate 368
- asICnrTime
 - ITime 588
- asInt
 - IOString 543
- asLongLong
 - IOString 544
- asPOINTL
 - IPoint 468
- asRECTL
 - IRectangle 480
- asSeconds
 - ITime 588
- IAssertionFailure 306
- assertParameter
 - IException 401
- assignment operator
 - See* operator =
- asSIZEL
 - ISize 501
- asString
 - IBase 309
 - IDatastoreBase 619
 - IDate 369
 - IPair 461
 - IPersistentObject 642
 - IRectangle 481
 - IOString 515
 - ITime 586
 - IVBase 599
- asUnsigned
 - IOString 544

- asUnsignedLong
 - IBitFlag 338
- asUnsignedLongLong
 - IStrng 544
- AT&T C++ Language System Release 1.2
 - overflow() 83
 - setbuf() 85
 - streambuf constructor 76
 - underflow() 86
- Datastore 648
- DatastoreBase 650
- DatastoreDB2 655
- DatastoreDB2Factory 660
- DatastoreFactory 661
- DatastoreODBC 662
- DatastoreODBCFactory 667
- attach() filebuf function 24
- attach() fstreambase function 28
- attachAsChild() tree function 248
- attachAsRoot() tree function 249
- attachSubtreeAsChild() tree function 248
- attachSubtreeAsRoot() tree function 249
- authentication
 - IDatastoreBase 619
- AuthenticationId
 - IDatastoreBase 622
- autoCommitId
 - IDatastoreDB2 630
 - IDatastoreODBC 638

B

- b2c
 - IStrng 511
- b2d
 - IStrng 511
- b2x
 - IStrng 511
- bad() ios function 43

- bag 131—135
- IBase 308
- IBase::Version 313
- base() streambuf function 78
- based-on concept in Collection Class Library
 - functions affected by
 - addAsFirst() 103
 - addAsLast() 103
 - addAsNext() 104
 - addAsPrevious() 104
 - addAtPostion() 105
 - addOrReplaceElementWithKey() 107
 - enqueue() 113
 - isBounded() 115
 - isFull() 115
 - locateOrAddElementWithKey() 119
 - maxNumberOfElements() 120
 - push() 122
 - general precondition 101, 102
- IBaseErrorInfo 314
- baseLibrary
 - IOException 401
- IBaseStream 320
- bitalloc() ios function 42
- IBitFlag 336
- blen() streambuf function 81
- IBoolean 96
- bottom
 - IRectangle 488
- bottomCenter
 - IRectangle 488
- bottomLeft
 - IRectangle 488
- bottomRight
 - IRectangle 488
- IBuffer 340
- IMemoryStream 422
- IStrng 551

built-in manipulators
 istream class 58
 ostream class 69

C

c_exception class 16—19
c2b
 IString 512
c2d
 IString 512
c2x
 IString 512
center
 IBuffer 341
 IDBCSBuffer 377
 IRectangle 489
 IString 516
centerAt
 IRectangle 482
centeredAt
 IRectangle 482
centerXCenterY
 IRectangle 486
centerXMaxY
 IRectangle 486
centerXMinY
 IRectangle 487
change
 IOString 290
 IBuffer 341
 IDBCSBuffer 377
 IString 516, 548
character conversion for numeric input 53
charLength
 IDBCSBuffer 388
charType
 IOString 294
 IBuffer 345
 IDBCSBuffer 381
 IString 536
checkAddition
 IBuffer 346
checkMultiplication
 IBuffer 347
childPositionAt() tree function 249
children of a tree node 238
CLASS_BASE_NAME 96
CLASS_NAME 96
className
 IBuffer 355
 IDBCSBuffer 388
clear() ios function 43
ICLibErrorInfo 358
CLibrary
 IException 402
close() function
 filebuf class 24
 fstreambase class 28
collection
 conditions for equality 100
collection abstract class 272
Collection Class Library
 abstract classes
 collection 272
 equality collection 273
 equality key collection 274
 equality key sorted collection 275
 equality sorted collection 276
 key collection 277
 key sorted collection 278
 ordered collection 279
 sequential collection 280
 sorted collection 281
 applicator classes 265
 concrete classes
 bag collection 131
 deque collection 136
 equality sequence collection 142
 heap collection 146
 key bag collection 149
 key set collection 155

Collection Class Library (*continued*)

concrete classes (*continued*)

- key sorted bag collection 162
- key sorted set collection 168
- map collection 176
- priority queue collection 184
- queue collection 188
- relation collection 192
- sequence collection 195
- set collection 201
- sorted bag collection 207
- sorted map collection 212
- sorted relation collection 219
- sorted set collection 224
- stack collection 231

- cursor classes 258

- pointer classes 266

- tree cursor classes 261

commit

- DatastoreBase 654

- IDatastore 613

- IDatastoreBase 620

- IDatastoreDB2 629

- IDatastoreODBC 637

compare

- IBuffer 341

- IDBCSBuffer 377

compare() function

- Collection Class Library 109

complex class 8—15

- constants 8

- conversion functions 14

- error handling 16

- input operator 12

- mathematical operators 10

- output operator 12

Complex Mathematics Library 8—19

complex_error() Complex Mathematics

- function 17

- conj() complex function 14

- conjugates of complex numbers 14

connect

- DatastoreBase 651

- IDatastore 612

- IDatastoreBase 617

- IDatastoreDB2 625

- IDatastoreODBC 633

connect_datastorename

- DatastoreBase 651

connect_defaults

- DatastoreBase 652

connect_string

- DatastoreDB2 655

- DatastoreODBC 662

connect_user

- DatastoreBase 652

connectedId

- IDatastoreBase 622

connectString

- IDatastoreDB2 627

- IDatastoreODBC 634

connectStringId

- IDatastoreDB2 630

- IDatastoreODBC 638

- constant applicator class 265

constructors

- Collection Class Library

- flat collections 99

- complex class 9

- cursor classes 259

- filebuf class 24

- flat collections 99

- fstream class 28

- ifstream class 30

- iostream class 48

- iostream_withassign class 48

- istream class 50

- istream_withassign class 59

- istrstream class 89

- constructors (*continued*)
 - ofstream class 32
 - ostream class 63
 - ostream_withassign class 70
 - ostrstream class 90
 - stdiobuf class 71
 - stdiostream class 72
 - strstream class 88
 - strstreambuf class 91
 - tree class 244
 - tree cursor class 261
- contains
 - IRectangle 490
- contains() Collection Class function 110
- containsAllFrom() Collection Class function 110
- containsAllKeysFrom() Collection Class function 110
- containsElementWithKey() Collection Class function 111
- contents
 - IBuffer 345
- context 362
 - IBaseStream 327
- conversion functions in complex class 14
- coord1
 - IPair 461
- coord2
 - IPair 461
- copy
 - IBuffer 342
 - IString 518
- copy constructors
 - flat collections 99
 - tree class 244
- copy() function
 - flat collections 111
 - tree class 249
- copySubtree() tree function 249

- cos() complex function 13
- cosh() complex function 13
- count
 - IContext 363
- create_object
 - DatastoreDB2Factory 660
 - DatastoreFactory 661
 - DatastoreODBCFactory 667
- create_object_defaults
 - DatastoreDB2Factory 660
 - DatastoreFactory 661
 - DatastoreODBCFactory 667
- Cursor
 - IObserverList::Cursor 450
- cursor classes 258—260

D

- d2b
 - IString 534
- d2c
 - IString 534
- d2x
 - IString 535
- IDAException 607
- data
 - IString 551
 - IStringTest 575
- IDatastore 610
- IDatastoreBase 616
- IDatastoreDB2 624
- datastoreName
 - IDatastoreBase 619
- DatastoreNameId
 - IDatastoreBase 622
- IDatastoreODBC 632
- IDate 365
- dayName
 - IDate 368, 370

- dayOfMonth
 - IDate 368
- dayOfWeek
 - IDate 368
- dayOfYear
 - IDate 369
- daysInMonth
 - IDate 371
- daysInYear
 - IDate 371
- IDBCSBuffer 376
- dbcsTable
 - IBuffer 356
- dbp() streambuf function 81
- ios::dec 37, 47
- defaultBuffer
 - IBuffer 345
 - IStrng 551
- defaultContext
 - IBaseStream 328
- degree of a tree node 238
- del
 - IPersistentObject 641
 - PersistentObject 669
- deleteId
 - IStandardNotifier 507
- deque 136—141
- dequeue() Collection Class function 112
- destructors
 - complex class 9
 - filebuf class 24
 - flat collections 99
 - istream class 89
 - ostream class 90
 - stdiobuf class 72
 - strstream class 88
 - strstreambuf class 92
 - tree class 245
- detach() function
 - filebuf class 24
 - fstreambase class 28
- IDeviceError 392
- difference
 - definition for bags 131
 - definition for flat collections 112
- differenceWith() Collection Class function 112
- disableInternationalization
 - IStrng 532
- disableNotification
 - INotifier 440
 - IStandardNotifier 504
- disableTrace
 - ITrace 593
- disableWriteLineNumber
 - ITrace 593
- disableWritePrefix
 - ITrace 593
- disconnect
 - DatastoreBase 652
 - IDatastore 612
 - IDatastoreBase 618
 - IDatastoreDB2 626
 - IDatastoreODBC 634
- disconnectedId
 - IDatastoreBase 622
- dispatchNotificationEvent
 - IObserver 446
- distanceFrom
 - IPair 464
- doallocate() function
 - streambuf class 83
 - strstreambuf class 92
- dotProduct
 - IPair 464
- driverPrompt
 - IDatastoreDB2 627
 - IDatastoreODBC 635

- driverPromptId
 - IDatastoreDB2 631
 - IDatastoreODBC 639
- dynamic mode 88, 91

E

- e (mathematical constant) 8
- eback() streambuf function 78
- ebuf() streambuf function 79
- egptr() streambuf function 79
- element() function
 - cursor classes 259
 - tree cursor class 262
- elementAt
 - IObserverList 448
- elementAt() function
 - flat collection classes 112—113
 - tree class 250
- elementAtPosition() Collection Class
 - function 113
- elementWithKey() Collection Class
 - function 113
- enable_sharemode_exclusive
 - Datastore 649
- enableAutoCommit
 - IDatastoreDB2 627
 - IDatastoreODBC 635
- enableInternationalization
 - IStrng 532
- enableNotification
 - INotifier 440
 - IStandardNotifier 505
- enableShareModeExclusive
 - IDatastore 613
- enableTrace
 - ITrace 593
- enableWriteLineNumber
 - ITrace 594
- enableWritePrefix
 - ITrace 594
- endl manipulator 69
- ends manipulator 69
- enqueue() Collection Class function 113
- eof() ios function 43
- epptr() streambuf function 79
- equal element 96
- equality collection abstract class 273
- equality key collection abstract class 274
- equality key sorted collection abstract class 275
- equality sequence 142—145
- equality sorted collection abstract class 276
- error
 - handling
 - by math.h for complex class 19
 - for complex class 16—19
 - ios error state 43
 - messages
 - Complex Mathematics Library 18
- errorAsString
 - IDAException 608
- errorCode
 - IDAException 608
- errorCodeGroup
 - IException 397
- errorId
 - IBaseErrorInfo 317
 - ICLibErrorInfo 359
 - IException 398
 - IGUIErrorInfo 415
 - ISystemErrorInfo 581
 - IXLibErrorInfo 603
- errorProvided
 - IDAException 608
- errorState
 - IDAException 608
- PersistentObject 668
- eventData
 - INotificationEvent 436

- examples
 - Collection Class Library
 - source files 2
 - heap 172
 - key bag 152
 - key set 160
 - key sorted bag 166
 - key sorted set 172
 - map 181
 - sequence 198
 - set 205
 - sorted map 217
 - sorted relation 217
 - sorted set 228
- Exception 394
- Exception::TraceFn 404
- ExceptionLocation 406
- exceptionLogged
 - Exception::TraceFn 405
- executeSQL
 - DatastoreBase 651
 - IDatastore 611
 - IDatastoreBase 617
 - IDatastoreDB2 625
 - IDatastoreODBC 633
- exp() complex function 12
- expandBy
 - IRectangle 482
- expandedBy
 - IRectangle 482

F

- fail() ios function 44
- fd() filebuf function 25
- filebuf class 23—26
- filebuf::openprot 31
- fileName
 - ExceptionLocation 406
- ifstream 408
- fill() ios function 39
- find
 - IContext 363
- findPhrase
 - IString 549
- firstElement() Collection Class function 114
- ios::fixed 38
- flags() ios function 40
- flush
 - IBaseStream 328
 - ifstream 409
- flush manipulator 69
- flush() ostream function 69
- fopen() library function 72
- forDisplay
 - IPersistentObject 642
- format flags in ios class
 - mutually exclusive flags 39
 - predefined 36—39
 - user-defined 42
- format state
 - fill character 39
 - flags 36—39
 - introduction 35
 - parameterized manipulators 60
 - precision 62
 - width variable 51, 52, 64
- format variables 35
- formatting
 - of input streams 51
 - of output streams 64
- freeze() function
 - ostream class 90
 - stringstream class 88
 - stringstreambuf class 93
- fromContents
 - IBuffer 346

fstream class 28—30
fstreambase class 27—28
functionName
 IExceptionLocation 406

G

gbump() streambuf function 82
gcount() istream function 57
get_connect_string
 DatastoreDB2 656
 DatastoreODBC 663
get_driver_prompt
 DatastoreDB2 656
 DatastoreODBC 663
get() istream function 55, 56
getline() istream function 56
getSqlca
 IDAException 609
good() ios function 44
gptr() streambuf function 79
IGUIErrorInfo 413

H

handleNotificationsFor
 IObserver 445
handleReadBufferEmpty
 IBaseStream 330
 IFileStream 411
 IMemoryStream 424
handleWriteBufferFull
 IBaseStream 330
 IFileStream 411
 IMemoryStream 425
hasChild() tree function 250
hasNotifierAttrChanged
 INotificationEvent 436
header files
 See chapters on individual classes

heap 146—148
 example of 172
height
 IRectangle 481
 ISize 501
height of a tree 238
ios::hex 37, 47
hours
 ITime 588

I

Note: Most classes beginning with an uppercase ‘I’ are listed under their second letter

I/O Stream Library

filebuf class 23
fstream class 28
fstreambase class 27
ifstream class 30
ios class 34
iostream class 48
iostream_withassign class 48
istream class 50
istream_withassign class 59
istrstream class 89
ofstream class 32
ostream class 63
ostream_withassign class 70
ostrstream class 90
parameterized manipulators 60
stdiobuf class 71
stdiostream class 72
streambuf class 74
strstream class 88
strstreambase class 87
strstreambuf class 91
IOString
 IOString 288
IAccessError
 IAccessError 304

- IAssertionFailure
 - IAssertionFailure 306
- IBaseErrorInfo
 - IBaseErrorInfo 317
- IBaseStream
 - IBaseStream 330
- IBitFlag
 - IBitFlag 339
- IBuffer
 - IBuffer 355
- ICLibErrorInfo
 - ICLibErrorInfo 359
- IContext
 - IContext 362
- IDatastore
 - IDatastore 611
- IDatastoreBase
 - IDatastoreBase 621
- IDatastoreDB2
 - IDatastoreDB2 625
- IDatastoreODBC
 - IDatastoreODBC 632
- IDate
 - IDate 367
- IDBCSBuffer
 - IDBCSBuffer 388
- IDeviceError
 - IDeviceError 392
- IException
 - IException 397
- IExceptionLocation
 - IExceptionLocation 407
- IFileStream
 - IFileStream 408
- ifstream class 30—32
- ignore() istream function 56
- IGUIErrorInfo
 - IGUIErrorInfo 414
- IInvalidParameter
 - IInvalidParameter 418
- IInvalidRequest
 - IInvalidRequest 420
- imag() complex function 15
- imaginary part of a complex number 9
- IMemoryStream
 - IMemoryStream 423
- IMessageText
 - IMessageText 426
- IMetaType
 - IMetaType 429
- implementation variant
 - See* chapters on individual Collection Classes
- in_avail() streambuf function 76
- includes
 - IRange 475
 - IString 535
- includesDBCS
 - IBuffer 344
 - IDBCSBuffer 380
 - IString 533
- includesMBCS
 - IBuffer 344
 - IDBCSBuffer 380
 - IString 533
- includesSBCS
 - IBuffer 344
 - IDBCSBuffer 380
 - IString 533
- indexOf
 - I0String 296
 - IBuffer 349
 - IDBCSBuffer 381
 - IString 525
- indexOfAnyBut
 - I0String 296
 - IBuffer 349
 - IDBCSBuffer 382
 - IString 526

- indexOfAnyOf
 - IOString 297
 - IBuffer 349
 - IDBCSBuffer 382
 - IString 527
- indexOfPhrase
 - IOString 300
 - IString 545
- indexOfWord
 - IOString 300
 - IString 545, 550
- initBuffer
 - IString 550
- initialize
 - IBuffer 355
 - IDate 374
 - ITime 589
- INotificationEvent
 - INotificationEvent 435
- INotifier
 - INotifier 440
- input operator
 - See* operator >>
- insert
 - IOString 292
 - IBuffer 342
 - IDBCSBuffer 378
 - IPointArray 471
 - IString 518, 548
- ios::internal 37
- internal classes
 - fstreambase class 27
 - strstreambase class 87
- intersection
 - bags 131
 - flat collections 114
- intersectionWith() Collection Class function 114
- intersects
 - IRectangle 490
- invalidate
 - IObserverList::Cursor 450
- invalidate() function
 - cursor classes 259
 - tree cursor class 262
- IInvalidParameter 418
- IInvalidRequest 420
- IObserver
 - IObserver 445
- IObserverList
 - IObserverList 447
- ios class 34—47
 - built-in manipulators 47
 - error checking 43
 - error state 43
 - format state
 - base conversion 37
 - buffer flushing 39
 - floating-point formatting 38
 - integral formatting 37
 - member functions 39
 - uppercase and lowercase 38
 - white space and padding 36
 - format state variables 35
- ios::app 29
- ios::ate 29
- ios::beg 68, 94
- ios::cur 68, 94
- ios::dec 37, 53, 66
- ios::end 68, 94
- ios::failbit 31, 32, 54
- ios::fixed 38
- ios::hex 37, 54, 66
- ios::in 30, 31, 94
- ios::internal 37
- ios::left 37
- ios::nocreate 30, 32
- ios::noreplace 30
- ios::oct 37, 53, 66

- ios::out 30, 94
- ios::right 37
- ios::scientific 38
- ios::showbase 37
- ios::showpoint 38
- ios::showpos 37
- ios::skipws 36
 - preventing looping 37
- ios::stdio 39, 71
- ios::trunc 30
- ios::unitbuf 39
- ios::uppercase 38
- ios::x_fill 61
- ios::x_prec 62
- ios::x_width 51, 52, 64
- iostream class 48—49
- iostream_withassign class 48—49
- IOutOfMemory
 - IOutOfMemory 453
- IOutOfSystemResource
 - IOutOfSystemResource 455
- IOutOfWindowResource
 - IOutOfWindowResource 457
- IPair
 - IPair 460
- IPersistentObject
 - IPersistentObject 644
- ipfx() istream function 51
- IPoint
 - IPoint 467
- IPointArray
 - IPointArray 471
- IPOManager
 - IPOManager 646
- IRange
 - IRange 474
- IRectangle
 - IRectangle 479
- IRefCounted
 - IRefCounted 493
- IReference
 - IReference 496
- IResourceExhausted
 - IResourceExhausted 498
- is_connected
 - DatastoreBase 652
- is_open() filebuf function 25
- is_sharemode_exclusive
 - Datastore 648
- isAbbrevFor
 - IString 552
- isAbbreviationFor
 - IString 535
- isAlphabetic
 - IBuffer 352
 - IDBCSBuffer 385
 - IString 541
- isAlphanumeric
 - IBuffer 352
 - IDBCSBuffer 385
 - IString 541
- isASCII
 - IBuffer 352
 - IDBCSBuffer 385
 - IString 541
- isAutoCommit
 - IDatastoreDB2 627
 - IDatastoreODBC 635
- isAvailable
 - IBaseErrorInfo 317
 - ICLibErrorInfo 359
 - IGUIErrorInfo 415
 - ISystemErrorInfo 581
 - IXLibErrorInfo 603
- isBinaryDigits
 - IString 542

- isBounded() Collection Class function 115
- isCharValid
 - IDBCSBuffer 389
- isConnected
 - IDatastoreBase 619
- isConnectedId
 - IDatastoreBase 623
- isControl
 - IBuffer 352
 - IDBCSBuffer 385
 - IString 542
- isDBCS
 - IBuffer 344
 - IDBCSBuffer 380
 - IString 533
- isDBCS1
 - IDBCSBuffer 390
- isDBCSLead
 - IBuffer 356
- isDefaultReadOnly
 - IPersistentObject 642
- isDigits
 - IBuffer 352
 - IDBCSBuffer 385
 - IString 542
- isEmpty
 - IObserverList 448
- isEmpty() function
 - flat collections 115
 - tree class 250
- isEnabledForNotification
 - INotifier 441
 - IStandardNotifier 505
- isFirst() Collection Class function 115
- isFull() Collection Class function 115
- isGraphics
 - IBuffer 352
 - IDBCSBuffer 385
 - IString 542
- isHexDigits
 - IBuffer 352
 - IDBCSBuffer 385
 - IString 542
- isInternationalized
 - IString 533
- ISize
 - ISize 500
- isLast() Collection Class function 115
- isLeaf() tree function 251
- isLeapYear
 - IDate 373
- isLike
 - IString 536, 552
- isLowerCase
 - IBuffer 353
 - IDBCSBuffer 386
 - IString 542
- isMBCS
 - IBuffer 345
 - IDBCSBuffer 381
 - IString 533
- isolationLevel
 - IDatastoreDB2 627
 - IDatastoreODBC 635
- isolationLevelId
 - IDatastoreDB2 631
 - IDatastoreODBC 639
- isPrevDBCS
 - IDBCSBuffer 390
- isPrintable
 - IBuffer 353
 - IDBCSBuffer 386
 - IString 542
- isPunctuation
 - IBuffer 353
 - IDBCSBuffer 386
 - IString 543
- isReadOnly
 - IPersistentObject 643

- isRecoverable
 - IOException 400
- isRetrievable
 - IPersistentObject 643
- isRoot() tree function 251
- isSBC
 - IDBCSBuffer 390
- isSBCS
 - IBuffer 345
 - IDBCSBuffer 381
 - IString 533
- isShareModeExclusive
 - IDatastore 612
- IStandardNotifier
 - IStandardNotifier 504
- isTraceEnabled
 - ITrace 593
- istream class 50—59
 - assignment operator 50
 - built-in manipulators 58
 - formatted input 51
 - input operator 52—55
 - input prefix function 51
 - unformatted input 55
- istream_withassign class 50—59
- IString
 - IString 513
- IStringParser
 - IStringParser 568
- IStringTest
 - IStringTest 574
- IStringTestMemberFn
 - IStringTestMemberFn 578
- istrstream class 87—90
- isUpperCase
 - IBuffer 353
 - IDBCSBuffer 386
 - IString 543
- isValid
 - IDate 373
 - IObserverList::Cursor 451
- isValid() function
 - cursor classes 259
 - tree cursor class 262
- isValidDBCS
 - IBuffer 345
 - IDBCSBuffer 381
 - IString 534
- isValidMBCS
 - IBuffer 345
 - IDBCSBuffer 381
 - IString 534
- isWhiteSpace
 - IBuffer 353
 - IDBCSBuffer 386
 - IString 543
- isWriteable
 - IBaseStream 328
 - IFileStream 409
- isWriteLineNumberEnabled
 - ITrace 594
- isWritePrefixEnabled
 - ITrace 594
- ISystemErrorInfo
 - ISystemErrorInfo 581
- iteration order 96, 241
- ITime
 - ITime 585
- ITrace
 - ITrace 592
- iword() ios function 42
- IXLibErrorInfo
 - IXLibErrorInfo 602

J

- julianDate
 - IDate 368

K

- key bag 149—154
- key collection abstract class 277
- key set 155—161
- key sorted bag 162—167
- key sorted collection abstract class 278
- key sorted set 168—175
- key() Collection Class function 115

L

- last-in, first-out behavior (LIFO) 231
- lastElement() Collection Class function 116
- lastIndexOf
 - IOString 297
 - IBuffer 350
 - IDBCSBuffer 382
 - IOString 539
- lastIndexOfAnyBut
 - IOString 298
 - IBuffer 350
 - IDBCSBuffer 383
 - IOString 539
- lastIndexOfAnyOf
 - IOString 299
 - IBuffer 350
 - IDBCSBuffer 383
 - IOString 540
- leaves of a tree 238
- ios::left 37
 - IRectangle 489
- leftCenter
 - IRectangle 489
- leftJustify
 - IBuffer 342
 - IDBCSBuffer 378
 - IOString 518
- length
 - IBuffer 346
 - IOString 537

- lengthOf
 - IOString 551
- lengthOfWord
 - IOString 545
- level of a tree node 238
- LIFO (last-in, first-out) behavior 231
- lineFrom
 - IOString 541
- lineNumber
 - IOExceptionLocation 406
- linked sequence 197
- locate() Collection Class function 116
- locateElementWithKey() Collection Class function 116
- locateFirst() Collection Class function 117
- locateLast() Collection Class function 117
- locateNext() Collection Class function 117
- locateNextElementWithKey() Collection Class function 118
- locateOrAdd() Collection Class function 118
- locateOrAddElementWithKey() Collection Class function 119
- locatePrevious() Collection Class function 120
- locationAtIndex
 - IOException 398
- locationCount
 - IOException 399
- log() complex function 12
- logData
 - IOException::TraceFn 405
- logExceptionData
 - IOException 399
- logicalEndOfStream
 - IBaseStream 328
 - IOFileStream 409
- lowerBound
 - IRange 475
- lowerCase
 - IBuffer 342
 - IDBCSBuffer 378

lowerCase (*continued*)

 IString 519

M

major

 IBase::Version 313

map 176—183

mathematical constants defined by complex
 class 8

mathematical functions for complex class 12

matherr() library function 19

maxCharLength

 IDBCSBuffer 389

maximum

 IPair 464

maxLong

 IString 553

maxNumberOfElements() Collection Class
 function 120

maxX

 IRectangle 481

maxXCenterY

 IRectangle 487

maxXMaxY

 IRectangle 487

maxXMinY

 IRectangle 487

maxY

 IRectangle 481

IMemoryStream 422

messageFile

 IBase 309

messageText 426

 IBase 310

IMetaType 429

IMetaTypeInfo 432

minimum

 IPair 464

minor

 IBase::Version 313

minutes

 ITime 588

minX

 IRectangle 481

minXCenterY

 IRectangle 487

minXMaxY

 IRectangle 487

minXMinY

 IRectangle 487

minY

 IRectangle 481

monthName

 IDate 371, 372

monthOfYear

 IDate 372

moveBy

 IRectangle 483

movedBy

 IRectangle 483

movedTo

 IRectangle 483

moveTo

 IRectangle 483

multiway tree class 240—244

N

n-ary tree class

 cursor class for 261

name

 IAccessError 305

 IAssertionFailure 307

 IDeviceError 393

 IException 401

 IInvalidParameter 419

 IInvalidRequest 421

 IMetaType 431

 IMetaTypeInfo 433

- name (*continued*)
 - IOutOfMemory 454
 - IOutOfSystemResource 456
 - IOutOfWindowResource 458
 - IResourceExhausted 499
- name data member of c_exception class 16
- newBuffer
 - IBuffer 347
- newCursor() function
 - flat collections 120
 - tree class 251
- next
 - IBuffer 346
 - IDBCSBuffer 381
- node of a tree 238
- norm() complex function 14
- notFound
 - IString 303
- INotificationEvent 435
- notificationId
 - INotificationEvent 436
- notifier 439
 - INotificationEvent 436
- notifyObservers
 - INotifier 441, 442
 - IObserverList 449
 - IStandardNotifier 505, 506
- now
 - ITime 586
- null
 - IBuffer 346
 - IString 553
- nullBuf
 - IString 551
- INumber Collection Class type 96
- numberOfChildren() tree function 251
- numberOfDifferentElements() Collection Class
 - function 120
- numberOfDifferentKeys() Collection Class
 - function 121

- numberOfElements
 - IObserverList 448
- numberOfElements() function
 - flat collections 121
 - tree class 251
- numberOfElementsWithKey() Collection Class
 - function 121
- numberOfLeaves() tree function 252
- numberOfOccurrences() Collection Class
 - function 121
- numberOfSubtreeElements() tree function 251
- numberOfSubtreeLeaves() tree function 252
- numberOfWords
 - IStringParser::SkipWords 571
- numWords
 - IString 545

O

- IObserver 444
- observerData
 - INotificationEvent 437
- observerList 447
 - INotifier 442
 - IStandardNotifier 506
- IObserverList::Cursor 450
- obsolete declarations
 - overflow() 83
 - streambuf constructor 76
- obsolete versions of functions
 - setbuf() 85
 - underflow() 86
- occurrencesOf
 - IString 299
 - IString 527, 550
- ios::oct 37, 47
- POFactory 670
- ofstream class 32—33
- open_mode enumeration 29

- open() function
 - filebuf class 25
 - fstream class 29
 - ifstream class 31
 - ofstream class 32
- operatingSystem
 - IException 402
- operator ~
 - IString 532
- operator -
 - complex class 10
 - IDate 372
 - IPair 461
 - ITime 587
- operator -=
 - complex class 11
 - IDate 372
 - IPair 463
 - ITime 587
- operator ->
 - IReference 497
- operator !
 - ios class 44
- operator !=
 - complex class 11
 - cursor classes 259
 - flat collections 100
 - IBitFlag 338
 - IDate 366
 - IMetaType 429
 - IMetaTypeInfo 433
 - IPair 460
 - IPointArray 470
 - IRectangle 478
 - ITime 584
 - tree cursor class 261
- operator /
 - complex class 11

- operator /=
 - complex class 11
 - IPair 463
- operator []
 - I0String 294
 - IPointArray 472
 - IString 537
- operator ()
 - ios class 44
- operator *
 - complex class 10
 - IReference 497
- operator *=
 - complex class 11
 - IPair 462
- operator &
 - IRectangle 486
 - IString 529
- operator &=
 - IRectangle 486
 - IString 529
- operator %=
 - IPair 462
- operator +
 - complex class 10
 - IDate 371
 - IString 529
 - ITime 587
- operator +=
 - complex class 11
 - IDate 371
 - IPair 463
 - IString 530
 - ITime 587
- operator <
 - IDate 366
 - IPair 460
 - ITime 585

- operator <<
 - char values 65
 - complex class 12
 - float and double values 66
 - integral values 66
 - IStringParser 563
 - ostream class 64—67
 - pointers to void 67
 - stream buffers 67
- operator <<=
 - IMetaType 430
- operator <=
 - IDate 366
 - IPair 460
 - ITime 585
- operator =
 - flat collections 100
 - IMessageText 428
 - IMetaType 430
 - INotificationEvent 436
 - ios class 35
 - iostream class 49
 - IPersistentObject 644
 - IPointArray 471
 - IPOManager 646
 - IReference 496
 - IStandardNotifier 504
 - istream_withassign class 59
 - IString 530
 - ostream_withassign class 70
 - tree class 245
- operator ==
 - complex class 11
 - cursor classes 260
 - flat collections 100
 - IBitFlag 338
 - IDate 366
 - IMetaType 429
 - IMetaTypeInfo 433
 - IPair 460
- operator == (*continued*)
 - IPersistentObject 644
 - IPointArray 470
 - IRectangle 479
 - ITime 585
 - tree cursor class 262
- operator >
 - IDate 366
 - IPair 460
 - ITime 585
- operator >=
 - IDate 366
 - IPair 460
 - ITime 585
- operator >>
 - arrays of char 52
 - complex class 12
 - float and double values 54
 - integral values 53
 - istream class 51—55
 - IStringParser 564, 565, 566, 567
 - pointers to char 52
 - references to char 53
 - streambuf objects 54
- operator >>=
 - IMetaType 431
- operator ^
 - IString 530
- operator ^=
 - IString 531
- operator |
 - IRectangle 486
 - IString 531
- operator |=
 - IRectangle 486
 - IString 532
- operator char *
 - IString 544
- operator const char *
 - IBaseErrorInfo 317

- operator const char * (*continued*)
 - ICLibErrorInfo 359
 - IGUIErrorInfo 415
 - IMessageText 428
 - ISystemErrorInfo 582
 - IXLibErrorInfo 603
- operator const void*
 - ios class 44
- operator delete
 - IBuffer 354
- operator new
 - IBuffer 354
- operator signed char *
 - IStrng 544
- operator T *
 - IReference 497
- operator unsigned char *
 - IStrng 544
- operator void*
 - ios class 44
- opfx() ostream function 64
- ordered collection abstract class 279
- osfx() ostream function 64, 71
- ostream class 63—70
 - built-in manipulators 69
 - formatted output 64
 - output operator 65
 - output prefix function 64
 - output suffix function 64
 - positioning 68
 - unformatted output 68
- ostream_withassign class 70
- ostrstream class 87—90
- other
 - IEException 402
- out_waiting() streambuf function 77
- IOutOfMemory 453
- IOutOfSystemResource 455
- IOutOfWindowResource 457
- output operator
 - See* operator <<
- output suffix function 64
- overflow
 - IBuffer 348
- overflow errors in complex class 14
- overflow() function
 - streambuf class 83
 - strstreambuf class 93
- overlayWith
 - I0String 293
 - IBuffer 342
 - IDBCSBuffer 378
 - IStrng 519, 549

P

- IPair 459
- parameterized manipulators
 - format state 60
 - introduction 60
 - resetiosflags() 61
 - setbase() 61
 - setfill() 61
 - setiosflags() 62
 - setprecision() 62
 - setw() 62
- parent in a tree 238
- pbackfail() streambuf function 84
- pbase() streambuf function 79
- pbump() streambuf function 82
- pcount() ostrstream function 90
- peek() istream function 58
- IPersistentObject 640
- physicalEndOfStream
 - IBaseStream 328
 - IFileStream 409
 - IMemoryStream 423

- pi (mathematical constant) 8
- IPoint 467
- IPointArray 470
- pointer class 266—268
- polar() complex function 15
- IPOManager 645
- pop() Collection Class function 121
- portability
 - publications 718
- IPosition Collection Class type 96
 - IBaseStream 329
 - IFileStream 409
 - IMemoryStream 423
- position() Collection Class function 121
- positioning property 96
- IPostorder Collection Class type 96
- postorder traversal of trees 238—239
- pow() complex function 13
- pptr() streambuf function 79
- precision() ios function 40
- IPreorder Collection Class type 96
- preorder traversal of trees 238—239
- presentationSystem
 - IException 402
- prevCharLength
 - IDBCSBuffer 389
- priority queue 184—187
 - dequeue() 112
 - enqueue() 113
- publications
 - related 718
- push() Collection Class function 122
- put() ostream function 68
- putback() istream function 58
- pwd() ios function 42

Q

- queue collection 188—191
 - dequeue() 112
 - enqueue() 113

R

- IRange 474
- rdbuf() function
 - fstream class 30
 - ifstream class 31
 - ios class 45
 - ofstream class 33
 - stdiostream class 72
 - strstreambase class 88
- rdstate() ios function 44
- read
 - IBaseStream 321, 325
- read() istream function 57
- readyId
 - IDatastore 615
 - IDatastoreDB2 631
 - IDatastoreODBC 639
- real part of a complex number 9
- real() complex function 15
- recoverable
 - IBase 311
- IRectangle 477
- IRefCounted 492
- IReference 495
- reference class
 - See* chapters on individual Collection Classes
- refresh
 - IPOManager 645
- related publications
 - portability 718
 - VisualAge for C++ 718
- relation 192—194
- remove
 - IString 293
 - IBuffer 343
 - IDBCSBuffer 379
 - IObserverList 448
 - IPointArray 472
 - IString 520

- remove() function
 - Collection Class Library 122
- removeAll
 - IObserverList 448
- removeAll() function
 - flat collections 123
 - tree class 252
- removeAllElementsWithKey() Collection Class
 - function 123
- removeAllObservers
 - INotifier 442
 - IStandardNotifier 506
- removeAllOccurrences() Collection Class
 - function 124
- removeAt
 - IObserverList 448
- removeAt() Collection Class function 124
- removeAtPosition() Collection Class
 - function 124
- removeElementWithKey() Collection Class
 - function 125
- removeFirst() Collection Class function 125
- removeLast() Collection Class function 126
- removeObserver
 - INotifier 442
 - IStandardNotifier 506
- removeRef
 - IBuffer 348
 - IRefCounted 493
- removeSubtree() tree function 252
- removeWords
 - IString 545
- replaceAt() function
 - flat collections 126
 - tree class 252
- replaceElementWithKey() Collection Class
 - function 126
- repositioning the get or put pointers 93
- reset
 - IContext 363
- resetiosflags() manipulator 61
- resize
 - IPointArray 472
- IResourceExhausted 498
- retrieve
 - IPersistentObject 641
 - PersistentObject 669
- retrieveAll
 - POFactory 670
- returned element 241
- retval 17
- reverse 127
 - IBuffer 343
 - IDBCSBuffer 379
 - IPointArray 472
 - IString 520
- reversed
 - IPointArray 472
- ios::right 37
 - IRectangle 489
- rightCenter
 - IRectangle 489
- rightJustify
 - IBuffer 343
 - IDBCSBuffer 379
 - IString 520
- rollback
 - DatastoreBase 654
 - IDatastore 614
 - IDatastoreBase 620
 - IDatastoreDB2 629
 - IDatastoreODBC 637
- root of a tree 238

S

- same key 96

- sbumpc() streambuf function 77
- scaleBy
 - IPair 463
 - IRectangle 483
- scaledBy
 - IPair 463
 - IRectangle 483
- ios::scientific 38
- seconds
 - ITime 588
- seek
 - IBaseStream 329
 - IFileStream 409
 - IMemoryStream 423
- seekg() function
 - istream class 57
 - ostream class 90
- seekoff() function
 - filebuf class 25
 - streambuf class 84
 - strstreambuf class 93
- seekp() ostream function 68
- seekpos() function
 - filebuf class 26
 - streambuf class 85
- seekRelative
 - IBaseStream 329
- select
 - IPOManager 646
 - POFactory 670
- sequence 195—200
- sequence as table 197
- sequential collection abstract class 280
- sequential collections
 - add() behavior with 101
 - conditions for equality 100
- set collection 201—206
- set_connect_string
 - DatastoreDB2 657
 - DatastoreODBC 664
- set_driver_prompt
 - DatastoreDB2 657
 - DatastoreODBC 664
- setAccessMode
 - IDatastoreDB2 628
 - IDatastoreODBC 635
- setAuthentication
 - IDatastore 613
 - IDatastoreBase 620
 - IDatastoreDB2 628
 - IDatastoreODBC 636
- setb() streambuf function 80
- setbase() manipulator 61
- setbuf() function
 - filebuf class 26
 - fstreambase class 28
 - streambuf class 85
 - strstreambuf class 94
- setBuffer
 - IString 551
- setConnectString
 - IDatastoreDB2 628
 - IDatastoreODBC 636
- setContext
 - IBaseStream 328
- setCoord1
 - IPair 461
- setCoord2
 - IPair 462
- setDatastoreName
 - IDatastore 613
 - IDatastoreBase 620
 - IDatastoreDB2 628
 - IDatastoreODBC 636
- setDefaultBuffer
 - IBuffer 348
- setDefaultText
 - IMessageText 428
- setDriverPrompt
 - IDatastoreDB2 628

- setDriverPrompt (*continued*)
 - IDatastoreODBC 636
- setErrorCodeGroup
 - IException 398
- setErrorId
 - IException 398
- setEventData
 - INotificationEvent 437
- setf() ios function 40
- setfill() manipulator 61
- setg() streambuf function 80
- setHeight
 - ISize 501
- setiosflags() manipulator 62
- setIsolationLevel
 - IDatastoreDB2 628
 - IDatastoreODBC 636
- setLogicalEndOfStream
 - IBaseStream 329
 - IFileStream 410
- setLowerBound
 - IRange 475
- setMessageFile
 - IBase 310
- setNotifierAttrChanged
 - INotificationEvent 437
- setObserverData
 - INotificationEvent 437
- setp() streambuf function 80
- setPhysicalEndOfStream
 - IBaseStream 329
 - IFileStream 411
 - IMemoryStream 425
- setprecision() manipulator 62
- setReadOnly
 - IPersistentObject 643
- setRetrievable
 - IPersistentObject 643
- setSeverity
 - IException 400
- setText
 - IException 400
- setToChild() function
 - tree class 253
 - tree cursor class 262
- setToFirst
 - IObserverList::Cursor 451
- setToFirst() function
 - cursor classes 260
 - flat collections 127
 - tree class 253
- setToFirstExistingChild() function
 - tree class 253
 - tree cursor class 262
- setToLast
 - IObserverList::Cursor 451
- setToLast() function
 - cursor classes 260
 - flat collections 127
 - tree class 254
- setToLastExistingChild() function
 - tree class 254
 - tree cursor class 263
- setToNext
 - IObserverList::Cursor 451
- setToNext() function
 - cursor classes 260
 - flat collections 128
 - tree class 254
- setToNextDifferentElement() Collection Class
 - function 128
- setToNextExistingChild() function
 - tree class 255
 - tree cursor class 263
- setToNextWithDifferentKey() Collection Class
 - function 128

- setToParent() function
 - tree class 255
 - tree cursor class 263
- setToPosition() Collection Class function 129
- setToPrevious
 - IObserverList::Cursor 451
- setToPrevious() function
 - cursor classes 260
 - flat collections 129
 - tree class 255
- setToPreviousExistingChild() function
 - tree class 256
 - tree cursor class 263
- setToRoot() function
 - tree class 256
 - tree cursor class 264
- setTraceFunction
 - IOException 399
- setUpperBound
 - IRange 475
- setUserName
 - IDatastore 613
 - IDatastoreBase 620
 - IDatastoreDB2 628
 - IDatastoreODBC 636
- setValue
 - IBitFlag 339
- setw() manipulator 62
- setWidth
 - ISize 501
- setX
 - IPoint 468
- setY
 - IPoint 468
- sgetc() streambuf function 77
- sgetn() streambuf function 77
- shareModeExclusiveId
 - IDatastore 615
- ios::showbase 37
- ios::showpoint 38
- ios::showpos 37
- shrinkBy
 - IRectangle 484
- shrunkBy
 - IRectangle 484
- sibling of a tree node 238
- sin() complex function 14
- sinh() complex function 14
- size 500
 - IPointArray 472
 - IRectangle 482
 - IString 538
- sizeBy
 - IRectangle 484
- sizedBy
 - IRectangle 485
- sizedTo
 - IRectangle 485
- sizeTo
 - IRectangle 485
- skip() ios function 41
- SkipWords
 - IStringParser::SkipWords 571
- ios::skipws 36
- snextc() streambuf function 77
- sort() Collection Class function 129
- sorted bag 207—211
- sorted collection abstract class 281
- sorted collections
 - add() behavior with 101
- sorted map 212—218
- sorted relation 219—223
 - example of 217
- sorted set 224—230
- source files for Collection Class Library
 - examples 2
- space
 - IString 546

- sputc() streambuf function 77
- sputc() streambuf function 78
- sputn() streambuf function 78
- sqrt() complex function 13
- square root of a complex number 13
- stack 231—236
- IStandardNotifier 503
- startBackwardsSearch
 - IBuffer 355
 - IDBCSBuffer 390
- startSearch
 - IBuffer 356
 - IDBCSBuffer 391
- stderr 71
- ios::stdio 39
- stdiobuf class 71—72
- stdiofile() stdiobuf function 72
- stdiostream class 72—73
- stdout 71
- stopHandlingNotificationsFor
 - IObserver 445
- stoss() streambuf function 78
- str() function
 - ostrstream class 90
 - strstream class 88
 - strstreambuf class 93
- stream buffer boundaries 78
- streambuf class 74—86
- IString 508
- IStringEnum 561
- IStringParser 562
- IStringParser::SkipWords 571
- IStringTest 573
- IStringTestMemberFn 577
- strip
 - IBuffer 343
 - IDBCSBuffer 379
 - IString 521, 549
- stripBlanks
 - IString 522
- stripLeading
 - IString 522
- stripLeadingBlanks
 - IString 523
- stripTrailing
 - IString 523
- stripTrailingBlanks
 - IString 524
- strstream class 87—90
- strstreambase class 88
- strstreambuf class 91—94
- subString
 - IString 295
 - IBuffer 351
 - IDBCSBuffer 384
 - IString 538
- subtree of a tree 238
- sync_with_stdio() ios function 45
- sync() function
 - filebuf class 26
 - istream class 58
 - streambuf class 86
- ISystemErrorInfo 580

T

- tellg() istream function 57
- tellp() ostream function 69
- template arguments
 - See* chapters on individual Collection Classes
- terminal of a tree 238
- terminate
 - IException 396
- test
 - IStringTest 574
 - IStringTestMemberFn 578
- text
 - IBaseErrorInfo 317
 - ICLibErrorInfo 360

- text (*continued*)
 - Exception 400
 - GUIErrorInfo 415
 - MessageText 428
 - SystemErrorInfo 582
 - XLibErrorInfo 603
- textCount
 - Exception 400
- this collection 97
- this tree 241
- threadId
 - ITrace 597
- throwCLibError
 - CLibErrorInfo 360
- throwError
 - BaseErrorInfo 317
- throwGUIError
 - GUIErrorInfo 416
- throwSystemError
 - SystemErrorInfo 582
- throwXLibError
 - XLibErrorInfo 603
- tie() ios function 46
- ITime 584
- today
 - IDate 368
- top
 - IRectangle 489
- top() Collection Class function 130
- topCenter
 - IRectangle 489
- topLeft
 - IRectangle 490
- topRight
 - IRectangle 490
- ITrace 590
- traceDestination
 - ITrace 594
- TraceFn
 - Exception::TraceFn 405

- transactedId
 - IDatastoreBase 623
- translate
 - IBuffer 344
 - IDBCSBuffer 380
 - IStrng 524, 549
- transpose
 - IPair 464
- tree 238—239
- tree class 240—244
- tree cursor class 261
- ITreeIterationOrder Collection Class type 96
- trigonometric functions for complex class 13
- type
 - IStrngTest 575
- type member of c_exception class 17
- TypeInfo
 - IMetaType 431

U

- unbuffered() streambuf function 82
- undefined cursor 96
- underflow() function
 - streambuf class 86
 - strstreambuf class 94
- unformatted input 55
- unformatted output 68
- union
 - bags 131
 - flat collections 130
- unionWith() Collection Class function 130
- unique collections
 - add() behavior with 101
 - conditions for equality 100
- ios::unitbuf 39
- unordered collections
 - locateNext() 117
 - locateNextElementWithKey() 118

- unrecoverable
 - IBase 311
- unsetf() ios function 41
- update
 - IPersistentObject 642
 - PersistentObject 669
- upperBound
 - IRange 475
- ios::uppercase 38
 - IBuffer 344
 - IDBCSBuffer 380
 - IString 525
- useCount
 - IBuffer 346
 - IRefCounted 493
- user-defined format flags in ios class 42
- userName
 - IDatastoreBase 619
- UserNameId
 - IDatastoreBase 623

V

- validate
 - IRectangle 491
- variant classes
 - See* chapters on individual Collection Classes
- IVBase 598
- version
 - IBase 309
- VisualAge for C++
 - publications 718

W

- width
 - IRectangle 482
 - ISize 501
- width() ios function 41

- word
 - IString 546
- wordIndexOfPhrase
 - IString 546
- words
 - IString 547
- write
 - IBaseStream 322, 326
 - IException::TraceFn 405
 - ITrace 594
- write() function
 - ostream class 68
- writeFormattedString
 - ITrace 596
- writeString
 - ITrace 596
- writeToFile
 - ITrace 595
- writeToQueue
 - ITrace 595
- writeToStandardError
 - ITrace 595
- writeToStandardOutput
 - ITrace 595

X

- x
 - IPoint 468
- x2b
 - IString 528
- x2c
 - IString 528
- x2d
 - IString 528
- xalloc() ios function 42
- IXLibErrorInfo 601

Y

y

IPoint 468

year

IDate 373

Z

zero

NSString 553

Communicating Your Comments to IBM

IBM VisualAge for C++ for Windows
Open Class Library Reference
Volume I
Version 3.5
Publication No. S33H-5039-00

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 for C++ for Windows
Open Class Library Reference
Volume I
Version 3.5
Publication No. S33H-5039-00

Overall, how satisfied are you with the information in this book?

| | Very Satisfied | Satisfied | Neutral | Dissatisfied | Very Dissatisfied |
|----------------------|--------------------------|--------------------------|--------------------------|--------------------------|--------------------------|
| Overall satisfaction | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |

How satisfied are you that the information in this book is:

| | Very Satisfied | Satisfied | Neutral | Dissatisfied | Very Dissatisfied |
|--------------------------|--------------------------|--------------------------|--------------------------|--------------------------|--------------------------|
| Accurate | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |
| Complete | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |
| Easy to find | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |
| Easy to understand | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |
| Well organized | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |
| Applicable to your tasks | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> | <input type="checkbox"/> |

Please tell us how we can improve this book:

Thank you for your responses. May we contact you? ☐ Yes ☐ No

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

Name

Address

Company or Organization

Phone No.

Readers' Comments — We'd Like to Hear from You
S33H-5039-00



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

S33H-5039-00

Cut or Fold
Along Line



Part Number: 33H5039
Program Number: 33H4979
33H4980

Printed in U.S.A.

S33H-5039-00



33H5039

