

OS/390



C/C++ IBM Open Class Library Reference

OS/390



C/C++ IBM Open Class Library Reference

Note!

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

Fourth Edition, September 1998

This edition applies to Version 2 Release 6 of OS/390 C/C++ (5647-A01) and to all subsequent releases and modifications until otherwise indicated in new editions or other updated documentation. Make sure that you use the correct edition for the level of the program listed above. Also, ensure that you apply all necessary PTFs for the program.

Technical changes in the text since the last release of this book are indicated by a vertical line (|) to the left of the change.

Order publications through your IBM representative or the IBM branch office serving your location. Publications are not stocked at the address below. The OS/390 C/C++ publications are available through the OS/390 Library page on the World Wide Web (<http://www.s390.ibm.com/os390/bkserv>).

IBM welcomes your comments. You can send your comments electronically to the network ID listed below. Be sure to include your entire network address if you wish a reply.

- Internet: torrcf@ca.ibm.com
- IBMLink: [toribm\(torrcf\)](#)
- IBM/PROFS: [torolab4\(torrcf\)](#)
- IBMMAIL: [ibmmail\(caibmwt9\)](#)

You can also send your comments by facsimile (attention: RCF coordinator) or you can use the Reader's Comment Form that is provided at the back of this publication. Refer to "Communicating Your Comments to IBM" for a description of the methods. This information immediately precedes the Reader's Comment Form at the back of this publication. You can also address your comments to:

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

If you send comments, include the title and order number of this book, and the page number or topic related to your comment.

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 1996, 1998. 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

Notices	xlvi
About This Book	li
About IBM OS/390 C/C++	lxi

Part 1. Complex Mathematics Library 1

Chapter 1. complex Class	3
Derivation	3
Header File	3
Members	3
Constants Defined in complex.h	3
Constructors for complex	4
Initializing complex Arrays	4
Mathematical Operators for complex	5
Addition	5
Subtraction	5
Negation	5
Multiplication	5
Division	5
Equality	5
Inequality	6
Mathematical Assignment Operators	6
Input and Output Operators for complex	6
Input Operator	6
Output Operator	6
Mathematical Functions for complex	7
exp	7
log	7
pow	7
sqrt	7
Trigonometric Functions for complex	8
cos	8
cosh	8
sin	8
sinh	8
Magnitude Functions for complex	8
abs	8
norm	8
Conversion Functions for complex	8
arg	9
conj	9
polar	9
real	9
imag	9
Chapter 2. c_exception Class	11
Derivation	11

Header File	11
Members	11
Constructor for <code>c_exception</code>	11
Data Members of <code>c_exception</code>	11
<code>arg1</code> , <code>arg2</code>	11
<code>name</code>	11
<code>retval</code>	12
<code>type</code>	12
Errors Handled by the Complex Mathematics Library	12
<code>complex_error</code>	12
Default Error-Handling Procedures	12

Part 2. I/O Stream Library 15

Chapter 3. <code>filebuf</code> Class	17
Derivation	17
Header File	17
Members	17
Public Members of <code>filebuf</code>	18
Constructors for <code>filebuf</code>	18
Destructor for <code>filebuf</code>	18
<code>attach</code>	18
<code>detach</code>	18
<code>close</code>	18
<code>fd</code>	19
<code>fp</code>	19
<code>is_open</code>	19
<code>open</code>	19
<code>seekoff</code>	19
<code>seekpos</code>	20
<code>setbuf</code>	20
<code>sync</code>	20
 Chapter 4. <code>fstream</code>, <code>ifstream</code>, and <code>ofstream</code> Classes	23
Derivation	23
Header File	23
Members	23
Public Members of <code>fstreambase</code>	23
<code>attach</code>	24
<code>close</code>	24
<code>detach</code>	24
<code>setbuf</code>	24
Public Members of <code>fstream</code>	24
Constructors for <code>fstream</code>	24
<code>open</code>	25
<code>rdbuf</code>	26
Public Members of <code>ifstream</code>	26
Constructors for <code>ifstream</code>	26
<code>open</code>	27
<code>rdbuf</code>	27
Public Members of <code>ofstream</code>	28
Constructors for <code>ofstream</code>	28
<code>open</code>	28

rdbuf	29
Chapter 5. ios Class	31
Derivation	31
Header File	31
Members	31
Constructors and Assignment Operator for ios	32
Format State Variables	32
x_fill	32
x_precision	32
x_width	33
Format State Flags	33
White Space and Padding	33
skipws	33
left	33
right	34
internal	34
Base Conversion	34
dec	34
oct	34
hex	34
showbase	34
Integral Formatting	34
showpos	34
Floating-Point Formatting	34
showpoint	34
scientific	34
fixed	35
Default Representation of Floating-Point Values	35
Uppercase and Lowercase	35
uppercase	35
Buffer Flushing	35
unitbuf	35
stdio	35
Mutually Exclusive Format Flags	35
Public Members of ios for the Format State	36
fill	36
flags	36
precision	36
setf	37
skip	37
unsetf	37
width	37
Public Members of ios for User-Defined Format Flags	38
bitalloc	38
iword	38
pword	38
xalloc	39
Public Members of ios for the Error State	39
bad	39
clear	39
eof	39
fail	40
good	40

rdstate	40
operator void*	40
operator!	40
Other Members of ios	41
rdbuf	41
sync_with_stdio	41
tie	41
Built-In Manipulators for ios	42
Chapter 6. istream and istream_withassign Classes	43
Derivation	43
Header File	43
Members	43
Public Members of istream and istream_withassign	43
Constructor for istream	43
Constructor for istream_withassign	43
Assignment Operator for istream_withassign	43
Chapter 7. ostream and ostream_withassign Classes	45
Derivation	45
Header File	45
Members	45
Constructors for ostream	45
Constructor for ostream	45
Input Prefix Function	45
Public Members of ostream for Formatted Output	46
Output Operator for Arrays of Characters	47
Output Operator for char	47
Output Operator for Other Integral Values	47
Output Operator for float and double Values	48
Output Operator for ostreambuf Objects	49
Public Members of ostream for Unformatted Output	49
get	49
get	49
get	50
get	50
getline	50
ignore	50
read	50
Public Members of ostream for Positioning	51
seekg	51
tellg	51
Other Public Members of ostream	51
gcount	51
peek	51
putback	52
sync	52
Built-In Manipulators for ostream	52
Public Members of ostream_withassign	53
Constructor for ostream_withassign	53
Assignment Operator for ostream_withassign	53
Chapter 8. Manipulators	55
Derivation	55

Header File	55
Members	55
Parameterized Manipulators for the Format State	55
resetiosflags	56
setbase	56
setfill	56
setiosflags	56
setprecision	57
setw	57
 Chapter 9. ostream and ostream_withassign Classes	59
Derivation	59
Header File	59
Members	59
Constructors for ostream	59
Constructor for ostream	59
Output Prefix and Suffix Functions	60
opfx	60
osfx	60
Public Members of ostream for Formatted Output	60
Output Operator for Arrays of Characters and char Values	61
Output Operator for Other Integral Values	62
Output Operator for float and double Values	62
Output Operator for Pointers to void	63
Output Operator for streambuf Objects	63
Public Members of ostream for Unformatted Output	63
put	63
write	63
Public Members of ostream for Positioning	64
seekp	64
tellp	64
Other Public Members of ostream	64
flush	64
Built-In Manipulators for ostream	64
Public Members of ostream_withassign	65
Constructor for ostream_withassign	65
Assignment Operator for ostream_withassign	65
 Chapter 10. stdiobuf and stdiostream Classes	67
Derivation	67
Header File	67
Members	67
Public Members of stdiobuf	67
Constructor for stdiobuf	67
Destructor for stdiobuf	68
stdiofile	68
Public Members of stdiostream	68
Constructor for stdiostream	68
rdbuf	68
Example of Using stdiostream	68
 Chapter 11. streambuf Class	71
Derivation	71
Header File	71

Members	71
streambuf Public and Protected Interfaces	71
What is the streambuf Public Interface?	72
What is the streambuf Protected Interface?	72
Public Members of the streambuf Public Interface	72
Constructors for streambuf	73
Destructor for streambuf	73
in_avail	73
out_waiting	73
sbumpc	73
sgetc	73
sgetn	74
snextc	74
sputbackc	74
sputc	74
sputn	74
stossc	74
Protected Functions That Return Pointers	75
base	75
eback	75
ebuf	75
egptr	75
epptr	75
gptr	75
pbase	75
pptr	76
Protected Functions That Set Pointers	76
setb	76
setg	76
setp	76
Other Nonvirtual Protected Member Functions	77
allocate	77
blen	77
dbp	77
gbump	78
pbump	78
unbuffered	78
Protected Virtual Member Functions	78
doallocate	79
overflow	79
pbackfail	79
seekoff	80
seekpos	80
setbuf	81
sync	81
underflow	81
 Chapter 12. strstream, istrstream, and ostrstream Classes	 83
Derivation	83
Header File	83
Members	83
Public Members of strstreambase	83
rdbuf	83
Public Members of strstream	84

Constructor for stringstream	84
Destructor for stringstream	84
str	84
Public Members of istream	84
Constructors for istream	84
Destructor for istream	85
Public Members of ostream	85
Constructors for ostream	85
Destructor for ostream	85
str	86
pcount	86
Chapter 13. stringstream Class	87
Derivation	87
Header File	87
Members	87
Public Members of stringstream	87
Constructors for stringstream	87
Destructor for stringstream	88
doallocate	88
freeze	88
overflow	89
str	89
seekoff	89
setbuf	90
underflow	90

Part 3. Flat Collection Classes 91

Chapter 14. Introduction to Flat Collections	93
Terms Used	93
Format of Class Descriptions	94
Required Functions	94
Types Defined for the Collection Class Library	95
Chapter 15. Flat Collection Member Functions	97
Constructor	97
Copy Constructor	97
Destructor	97
operator!=	97
operator=	98
operator==	98
add	98
addAllFrom	99
addAsFirst	100
addAsLast	100
addAsNext	101
addAsPrevious	101
addAtPosition	101
addDifference	102
addIntersection	103
addOrReplaceElementWithKey	103
addUnion	104

allElementsDo	105
allElementsDo	105
any	106
compare	106
contains	106
containsAllFrom	106
containsAllKeysFrom	107
containsElementWithKey	107
copy	107
deque	108
differenceWith	108
disableNotification	108
elementAt	108
elementAtPosition	109
elementWithKey	109
enableNotification	109
enqueue	109
first	110
intersectionWith	110
isBounded	110
isEmpty	110
isEnabledForNotification	111
isFirstAt	111
isFull	111
isLastAt	111
key	111
last	111
locate	112
locateElementWithKey	112
locateFirst	112
locateLast	112
locateNext	113
locateNextElementWithKey	113
locateOrAdd	113
locateOrAddElementWithKey	114
locatePrevious	115
maxNumberOfElements	115
newCursor	115
notifier	115
notifyObservers	116
numberOfDifferentElements	116
numberOfDifferentKeys	116
numberOfElements	116
numberOfElementsWithKey	116
numberOfOccurrences	116
pop	116
positionAt	117
push	117
remove	117
removeAll	118
removeAll	118
removeAllElementsWithKey	118
removeAllOccurrences	118
removeAt	119

removeAtPosition	119
removeElementWithKey	119
removeFirst	120
removeLast	120
replaceAt	120
replaceElementWithKey	121
reverse	121
setToFirst	121
setToLast	121
setToNext	122
setToNextDifferentElement	122
setToNextWithDifferentKey	122
setToPosition	123
setToPrevious	123
sort	123
top	124
unionWith	124
Chapter 16. Bag	125
Derivation	125
Variants and Header Files	125
Members	126
Template Arguments and Required Functions	126
Bag	126
Bag as Hash Table	127
Bag as List	127
Bag as Table	127
Bag as Diluted Table	128
Abstract Class	128
Coding Example for Bag	128
Chapter 17. Deque	131
Derivation	131
Variants and Header Files	131
Members	132
Template Arguments and Required Functions	132
Deque	132
Deque as List	133
Deque as Table	133
Deque as Diluted Table	133
Abstract Class	133
Coding Example for Deque	134
Chapter 18. Equality Sequence	137
Derivation	137
Variants and Header Files	137
Members	138
Template Arguments and Required Functions	139
Equality Sequence	139
Equality Sequence as List	139
Equality Sequence as Table	139
Equality Sequence as Diluted Table	139
Abstract Class	140

Chapter 19. Heap	141
Derivation	141
Variants and Header Files	141
Members	142
Template Arguments and Required Functions	142
Heap	142
Heap as List	142
Heap as Table	143
Heap as Diluted Table	143
Abstract Class	143
Coding Example for Heap	143
 Chapter 20. Key Bag	 145
Derivation	145
Variants and Header Files	145
Members	146
Template Arguments and Required Functions	146
Key Bag	146
Key Bag as Hash Table	147
Abstract Class	147
Coding Example for Key Bag	147
 Chapter 21. Key Set	 151
Derivation	151
Variants and Header Files	151
Members	152
Template Arguments and Required Functions	152
Key Set	152
Key Set as AVL Tree	153
Key Set as B* Tree	153
Key Set as Hash Table	153
Key Set as List	154
Key Set as Table	154
Key Set as Diluted Table	154
Abstract Class	155
Coding Example for Key Set	155
 Chapter 22. Key Sorted Bag	 157
Derivation	157
Variants and Header Files	157
Members	158
Template Arguments and Required Functions	158
Key Sorted Bag	158
Key Sorted Bag as List	159
Key Sorted Bag as Table	159
Key Sorted Bag as Diluted Table	159
Abstract Class	160
Coding Example for Key Sorted Bag	160
 Chapter 23. Key Sorted Set	 163
Derivation	163
Variants and Header Files	163
Members	164
Template Arguments and Required Functions	164

Key Sorted Set	164
Key Sorted Set as AVL Tree	165
Key Sorted Set as B* Tree	165
Key Sorted Set as List	165
Key Sorted Set as Table	166
Key Sorted Set as Diluted Table	166
Abstract Class	166
Coding Example for Key Sorted Set	167
Chapter 24. Map	171
Derivation	171
Variants and Header Files	171
Members	172
Template Arguments and Required Functions	173
Map	173
Map as AVL Tree	173
Map as B* Tree	173
Map as List	174
Map as Table	174
Map as Diluted Table	174
Map as Hash Table	175
Abstract Class	175
Coding Example for Map	176
Chapter 25. Priority Queue	179
Derivation	179
Variants and Header Files	179
Members	180
Template Arguments and Required Functions	180
Priority Queue	180
Priority Queue as List	181
Priority Queue as Table	181
Priority Queue as Diluted Table	182
Abstract Class	182
Chapter 26. Queue	183
Derivation	183
Variants and Header Files	183
Members	184
Template Arguments and Required Functions	184
Queue	184
Queue as List	185
Queue as Table	185
Queue as Diluted Table	185
Abstract Class	186
Chapter 27. Relation	187
Derivation	187
Variants and Header Files	187
Members	188
Template Arguments and Required Functions	188
Abstract Class	189
Chapter 28. Sequence	191

Derivation	191
Variants and Header Files	191
Members	192
Template Arguments and Required Functions	192
Sequence	192
Sequence as List	193
Sequence as Table	193
Sequence as Diluted Table	193
Abstract Class	193
Coding Example for Sequence	194
Chapter 29. Set	197
Derivation	197
Variants and Header Files	197
Members	198
Template Arguments and Required Functions	199
Set	199
Set as AVL Tree	199
Set as B* Tree	199
Set as List	199
Set as Table	200
Set as Diluted Table	200
Set as Hash Table	200
Abstract Class	201
Coding Example for Set	201
Chapter 30. Sorted Bag	203
Derivation	203
Variants and Header Files	203
Members	204
Template Arguments and Required Functions	204
Sorted Bag	204
Sorted Bag as List	205
Sorted Bag as Table	205
Sorted Bag as Diluted Table	205
Abstract Class	205
Chapter 31. Sorted Map	207
Derivation	207
Variants and Header Files	207
Members	208
Template Arguments and Required Functions	209
Sorted Map	209
Sorted Map as AVL Tree	209
Sorted Map as B* Tree	209
Sorted Map as List	210
Sorted Map as Table	210
Sorted Map as Diluted Table	210
Abstract Class	211
Coding Example for Sorted Map	211
Chapter 32. Sorted Relation	219
Derivation	219
Variants and Header Files	219

Members	220
Template Arguments and Required Functions	221
Sorted Relation	221
Sorted Relation as List	221
Sorted Relation as Table	221
Sorted Relation as Diluted Table	222
Abstract Class	222
Coding Example for Sorted Relation	222
Chapter 33. Sorted Set	223
Derivation	223
Variants and Header Files	223
Members	224
Template Arguments and Required Functions	225
Sorted Set	225
Sorted Set as AVL Tree	225
Sorted Set as B* Tree	225
Sorted Set as List	225
Sorted Set as Table	226
Sorted Set as Diluted Table	226
Abstract Class	226
Coding Example for Sorted Set	226
Chapter 34. Stack	229
Derivation	229
Variants and Header Files	229
Members	230
Template Arguments and Required Functions	230
Stack	230
Stack as List	231
Stack as Table	231
Stack as Diluted Table	231
Abstract Class	231
Coding Example for Stack	232

Part 4. Tree Collection Classes 235

Chapter 35. Introduction to Trees	237
Defining the Traversal Order of Tree Elements	237
IPreorder	237
IPostorder	238
Chapter 36. Multiway Tree	239
Derivation	239
Variants and Header Files	239
Members	239
Template Arguments and Required Functions	239
Terms Used	240
Coding Example for Multiway Tree	240
Tree Functions	244
Constructor	244
Copy Constructor	244
Destructor	244

operator=	244
addAsChild	245
addAsRoot	245
allElementsDo, allSubtreeElementsDo	245
allElementsDo, allSubtreeElementsDo	246
attachAsChild, attachSubtreeAsChild	247
attachAsRoot, attachSubtreeAsRoot	247
childPositionAt	248
copy, copySubtree	248
elementAt	248
hasChild	249
isEmpty	249
isLeaf	249
isRoot	249
newCursor	249
numberOfChildren	250
numberOfElements, numberOfSubtreeElements	250
numberOfLeaves, numberOfSubtreeLeaves	250
removeAll, removeSubtree	250
replaceAt	251
setToChild	251
setToFirst	251
setToFirstExistingChild	251
setToLast	252
setToLastExistingChild	252
setToNext	252
setToNextExistingChild	252
setToParent	253
setToPrevious	253
setToPreviousExistingChild	253
setToRoot	253

Part 5. Auxiliary Collection Classes 255

Chapter 37. Cursor	257
Header File	257
Members	257
Public Member Functions	258
Constructor	258
copy	258
isValid	258
invalidate	258
element	258
operator!=	258
operator==	258
setToFirst	259
setToLast	259
setToNext	259
setToPrevious	259
Chapter 38. Tree Cursor	261
Header Files	261
Members	261

Public Members of Tree Cursor	261
Constructor	261
operator!=	261
operator==	261
element	262
isValid	262
invalidate	262
setToChild	262
setToFirstExistingChild	262
setToLastExistingChild	262
setToNextExistingChild	263
setToParent	263
setToPreviousExistingChild	263
setToRoot	263
Chapter 39. Applicator and Constant Applicator Classes	265
Derivation	265
Header File	265
Members	265
applyTo	265
Chapter 40. Pointer Classes	267
Members	267
Constructors	267
Constructors from a Given C++ Pointer	267
Copy Constructors from a Given Collection Class Pointer	268
Destructors	268
operator*	268
Conversion operator	268
operator->	268
operator=	268
operator==	268
Coding Example for Managed Element Pointer	269
Chapter 41. Collection Event Data	273
Derivation	273
Inherited By	273
Header File	273
Class Name	273
Members	273
Members	273
cursor	273
element	273
Chapter 42. Collection Guard	275
Derivation	275
Inherited By	275
Header File	275
Class Name	275
Members	275
Members	275
Constructor	275
Destructor	275

Chapter 43. Restricted Access Collection Guard	277
Derivation	277
Inherited By	277
Header File	277
Class Name	277
Members	277
Members	277
Constructor	277
Destructor	277
 Chapter 44. Tree Collection Guard	279
Derivation	279
Inherited By	279
Header File	279
Class Name	279
Members	279
Members	279
Constructor	279
Destructor	279

Part 6. Abstract Collection Classes	281
--	-----

Chapter 45. Collection	283
Derivation	283
Header File	283
Members	283
 Chapter 46. Equality Collection	285
Derivation	285
Header File	285
Members	285
 Chapter 47. Equality Key Collection	287
Derivation	287
Header File	287
Members	287
 Chapter 48. Equality Key Sorted Collection	289
Derivation	289
Header File	289
Members	289
 Chapter 49. Equality Sorted Collection	291
Derivation	291
Header File	291
Members	291
 Chapter 50. Key Collection	293
Derivation	293
Header File	293
Members	293
 Chapter 51. Key Sorted Collection	295

Derivation	295
Header File	295
Members	295
Chapter 52. Ordered Collection	297
Derivation	297
Header File	297
Members	297
Chapter 53. Sequential Collection	299
Derivation	299
Header File	299
Members	299
Chapter 54. Sorted Collection	301
Derivation	301
Header File	301
Members	301
Chapter 55. Restricted Access Collection	303
Derivation	303
Inherited By	303
Header File	303
Class Name	303
Members	303
<hr/>	
Part 7. Application Support Class Library	305
Chapter 56. Base Classes	311
IBase	311
Derivation	311
Inherited By	311
Header File	311
Nested Classes	311
Public Members	311
asDebugInfo	311
asString	311
messageFile	312
messageText	312
operator<<	312
setMessageFile	313
version	313
Enumerations	313
BooleanConstants	313
IVBase	313
Derivation	313
Header File	313
Public Members	314
asDebugInfo	314
asString	314
operator<<	314
Inherited Public Members	314

Chapter 57. Buffer Classes	315
IBuffer	315
Derivation	315
Header File	315
Constructors	315
Public Members	315
addRef	315
asDebugInfo	315
center	315
change	315
charType	316
checkAddition	316
checkMultiplication	316
compare	316
contents	316
copy	316
defaultBuffer	316
fromContents	317
includesDBCS	317
includesMBCS	317
includesSBCS	317
indexOf	317
indexOfAnyBut	317
indexOfAnyOf	317
insert	318
isAlphabetic	318
isAlphanumeric	318
isASCII	318
isControl	318
isDBCS	318
isDigits	318
isGraphics	318
isHexDigits	319
isLowerCase	319
isMBCS	319
isPrintable	319
isPunctuation	319
isSBCS	319
isUpperCase	319
isValidDBCS	320
isValidMBCS	320
isWhiteSpace	320
lastIndexOf	320
lastIndexOfAnyBut	320
lastIndexOfAnyOf	321
leftJustify	321
length	321
lowerCase	321
newBuffer	321
next	322
null	322
overflow	322
overlayWith	322
remove	322

removeRef	322
reverse	323
rightJustify	323
setDefaultBuffer	323
strip	323
subString	323
translate	324
upperCase	324
useCount	324
Inherited Public Members	324
Protected Members	324
allocate	324
className	324
initialize	324
operator delete	324
operator new	324
startBackwardsSearch	325
startSearch	325
Nested Type Definitions	325
Comparison	325
IDBCSBuffer	325
Derivation	325
Header File	326
Constructors	326
Public Members	326
allocate	326
center	326
charType	326
includesDBCS	326
includesMBCS	326
includesSBCS	326
indexOf	327
indexOfAnyBut	327
indexOfAnyOf	327
insert	327
isDBCS	327
isMBCS	327
isSBCS	327
isValidDBCS	327
isValidMBCS	328
lastIndexOf	328
lastIndexOfAnyBut	328
lastIndexOfAnyOf	328
leftJustify	328
lowerCase	329
next	329
overlayWith	329
remove	329
reverse	329
rightJustify	329
strip	329
subString	330
translate	330
upperCase	330

Inherited Public Members	330
Protected Members	331
charLength	331
className	331
isCharValid	331
isDBCS1	332
isPrevDBCS	332
isSBC	332
maxCharLength	332
prevCharLength	332
startBackwardsSearch	333
startSearch	333
Inherited Protected Members	333
Chapter 58. IDate Class	335
Derivation	335
Header File	335
Constructors	335
Public Members	336
asCDate	336
asString	336
dayName	337
dayOfMonth	337
dayOfWeek	337
dayOfYear	337
daysInMonth	337
daysInYear	337
isLeapYear	337
isValid	337
julianDate	338
monthName	338
monthOfYear	338
operator!=	338
operator+	338
operator+=	338
operator-	338
operator-=	339
operator<	339
operator<<	339
operator<=	339
operator==	339
operator>	339
operator>=	339
today	339
year	339
Inherited Public Members	340
Protected Members	340
initialize	340
Enumerations	340
DayOfWeek	340
Month	340
YearFormat	340
Chapter 59. Exception Classes	341

Exception	341
Derivation	341
Header File	341
Nested Classes	343
Constructors	343
Error Code	343
errorCodeGroup	343
setErrorCodeGroup	343
Public Members	343
addLocation	343
appendText	344
assertParameter	344
errorId	344
isRecoverable	344
locationAtIndex	344
locationCount	345
logExceptionData	345
name	345
setErrorId	345
setSeverity	345
setText	345
setTraceFunction	345
terminate	346
text	346
textCount	346
Enumerations	346
Severity	346
Public Data	347
Error Code	347
baseLibrary	347
CLibrary	347
operatingSystem	347
other	347
presentationSystem	347
Nested Classes	347
Nested Type Definitions	348
Severity	348
ErrorCodeGroup	348
Protected Members	348
Constructors	348
TraceFn	348
Tracing	348
exceptionLogged	348
IAccessError	349
Derivation	349
Header File	349
Constructors	349
Public Members	349
name	349
Inherited Public Members	350
IAssertionFailure	350
Derivation	350
Header File	350
Constructors	350

Public Members	350
name	350
Inherited Public Members	351
ICLibErrorInfo	351
Derivation	351
Header File	351
Constructors	351
Public Members	352
errorId	352
isAvailable	352
operator const char *	352
text	352
throwCLibError	352
Inherited Public Members	353
IDeviceError	353
Derivation	353
Header File	353
Constructors	353
Public Members	353
name	353
Inherited Public Members	354
IBaseErrorInfo	354
Derivation	354
Header File	354
Public Members	355
errorId	355
isAvailable	355
operator const char *	355
text	355
throwError	356
Inherited Public Members	356
Enumerations	356
ExceptionType	356
IException::TraceFn	357
Derivation	357
Header File	357
Public Members	357
write	357
IExceptionLocation	358
Derivation	358
Header File	358
Constructors	358
Public Members	358
fileName	358
functionName	359
lineNumber	359
IGUIErrorInfo	359
Derivation	359
Header File	359
Constructors	360
Public Members	360
errorId	360
isAvailable	361
operator const char *	361

text	361
throwError	361
throwGUIError	361
Inherited Public Members	362
InvalidParameter	362
Derivation	362
Header File	362
Constructors	362
Public Members	363
name	363
Inherited Public Members	363
InvalidRequest	363
Derivation	363
Header File	363
Constructors	363
Public Members	364
name	364
Inherited Public Members	364
IMessageText	364
Derivation	364
Header File	364
Constructors	364
Public Members	366
operator=	366
operator const char *	366
setDefaultText	366
text	366
IOutOfMemory	366
Derivation	366
Header File	366
Constructors	367
Public Members	367
name	367
Inherited Public Members	367
IOutOfSystemResource	367
Derivation	367
Header File	367
Constructors	368
Public Members	368
name	368
Inherited Public Members	368
IOutOfWindowResource	368
Derivation	368
Header File	369
Constructors	369
Public Members	369
name	369
Inherited Public Members	369
IResourceExhausted	369
Derivation	370
Header File	370
Constructors	370
Public Members	370
name	370

Inherited Public Members	371
ISystemErrorInfo	371
Derivation	371
Header File	371
Constructors	372
Public Members	372
errorId	372
isAvailable	372
operator const char *	372
text	372
throwSystemError	372
Inherited Public Members	373
IXLibErrorInfo	373
Derivation	373
Header File	373
Constructors	374
Public Members	374
errorId	374
isAvailable	374
operator const char *	375
text	375
throwXLibError	375
Inherited Public Members	375
Chapter 60. String Classes	377
IString	377
Derivation	377
Header File	377
Constructors	378
Public Members	379
asDebugInfo	379
asDouble	379
asInt	379
asLongLong	380
asString	380
asUnsigned	380
asUnsignedLongLong	380
b2c	380
b2d	380
b2x	380
c2b	380
c2d	381
c2x	381
center	381
change	381
charType	382
copy	382
d2b	382
d2c	382
d2x	383
disableInternationalization	383
enableInternationalization	383
includes	383
includesDBCS	383

includesMBCS	383
includesSBCS	383
indexOf	383
indexOfAnyBut	384
indexOfAnyOf	384
indexOfPhrase	384
indexOfWord	384
insert	384
isAbbreviationFor	385
isAlphabetic	385
isAlphanumeric	385
isASCII	385
isBinaryDigits	385
isControl	386
isDBCS	386
isDigits	386
isGraphics	386
isHexDigits	386
isInternationalized	386
isLike	386
isLowerCase	387
isMBCS	387
isPrintable	387
isPunctuation	387
isSBCS	387
isUpperCase	388
isValidDBCS	388
isValidMBCS	388
isWhiteSpace	388
lastIndexOf	388
lastIndexOfAnyBut	389
lastIndexOfAnyOf	389
leftJustify	389
length	389
lengthOfWord	390
lineFrom	390
lowerCase	390
numWords	390
occurrencesOf	390
operator!=	390
operator&	391
operator&=	391
operator+	391
operator+=	392
operator<	392
operator<<	392
operator<=	392
operator=	393
operator==	393
operator>	393
operator>=	394
operator>>	394
operator char *	394
operator signed char *	394

operator unsigned char *	394
operator[]	395
operator^	395
operator^=	395
operator	395
operator =	396
operator~	396
overlayWith	396
remove	396
removeWords	397
reverse	397
rightJustify	397
size	397
space	397
strip	397
stripBlanks	398
stripLeading	398
stripLeadingBlanks	398
stripTrailing	398
stripTrailingBlanks	399
subString	399
translate	399
upperCase	400
word	400
wordIndexOfPhrase	400
words	400
x2b	400
x2c	400
x2d	401
Inherited Public Members	401
Protected Members	401
applyBitOp	401
buffer	401
change	401
data	402
defaultBuffer	402
findPhrase	402
indexOfWord	402
initBuffer	402
insert	403
isAbbrevFor	403
isLike	403
lengthOf	404
occurrencesOf	404
overlayWith	404
setBuffer	404
strip	404
translate	405
IOString	405
Derivation	405
Header File	405
Constructors	406
Public Members	407
change	407

charType	408
indexOf	408
indexOfAnyBut	408
indexOfAnyOf	408
indexOfPhrase	408
indexOfWord	408
insert	408
lastIndexOf	409
lastIndexOfAnyBut	409
lastIndexOfAnyOf	409
occurrencesOf	410
operator[]	410
overlayWith	410
remove	410
subString	411
Inherited Public Members	411
Protected Members	411
adjustArg	411
adjustResult	411
IStringEnum	412
Derivation	412
Header File	412
Enumerations	412
CharType	412
StripMode	413
IStringParser	413
Derivation	413
Header File	413
Nested Classes	414
Constructors	414
Public Members	415
operator<<	415
operator>>	415
Inherited Public Members	416
Enumerations	416
Command	416
IStringParser::SkipWords	416
Derivation	416
Header File	416
Constructors	416
Public Members	417
numberOfWords	417
Inherited Public Members	417
IStringTest	417
Derivation	417
Header File	417
Constructors	417
Public Members	418
test	418
Inherited Public Members	418
Enumerations	418
FnType	418
IStringTestMemberFn	418
Derivation	418

Header File	418
Constructors	419
Public Members	419
test	419
Inherited Public Members	419
Chapter 61. IApplication	421
Derivation	421
Inherited By	421
Header File	421
Members	421
Public Functions	421
Diagnostics	421
asDebugInfo	422
asString	422
Priority	422
adjustPriority	422
setPriority	423
Process Information	423
current	423
currentPID	424
id	424
Inherited Public Functions	424
Protected Functions	424
Constructors	424
IApplication	424
~IApplication	424
Setting Process Information	424
setId	425
Inherited Protected Data	425
PriorityClass	425
Chapter 62. Decimal Classes	427
IBinaryCodedDecimal	427
Derivation	427
Inherited By	427
Header File	427
Members	427
Constructors	427
IBinaryCodedDecimal	427
Public Members	430
Comparisons	430
operator !=	430
operator <	430
operator <=	431
operator ==	431
operator >	431
operator >=	431
Manipulations	431
operator !	431
operator *=	431
operator +	431
operator ++	431
operator +=	432

operator -	432
operator --	432
operator -=	432
operator /=	433
operator =	433
Queries	434
cData	434
digitsOf	434
isNegative	434
isPositive	434
precisionOf	435
Type Conversions	435
asDouble	435
asLong	435
asLongLong	435
asString	435
Protected Functions	435
Streaming	435
readFromStream	435
writeToStream	436
Decimal	436
Derivation	436
Inherited By	436
Header File	436
Constructors	436
Decimal	436
Public Members	438
Comparisons	438
operator !=	438
operator <	438
operator <=	438
operator ==	438
operator >	438
operator >=	438
Manipulations	438
operator !	438
operator *	439
operator *=	439
operator +	439
operator ++	439
operator +=	439
operator -	439
operator --	439
operator -=	439
operator /	439
operator /=	440
operator =	440
Streaming	440
operator <<	440
operator >>	440
Queries	440
cData	440
digitsOf	440
isNegative	441

isPositive	441
precisionOf	441
Type Conversions	441
asBCD	441
asString	441
Chapter 63. ICurrentApplication	443
Derivation	443
Inherited By	443
Header File	443
Members	443
Public Functions	443
Arguments	443
argc	443
argv	444
setArgs	444
Diagnostics	444
asDebugInfo	444
Starting and Stopping	444
exit	444
run	445
Inherited Public Functions	445
Protected Functions	445
Constructors	445
ICurrentApplication	445
~ICurrentApplication	445
Process Information	445
pib	446
Inherited Protected Functions	446
Inherited Protected Data	446
Chapter 64. ICurrentThread	447
Derivation	447
Inherited By	447
Header File	447
Members	447
Public Functions	448
Current Thread Information	448
handle	448
id	448
Current Thread Support	448
exit	448
isTopLevelShell	448
isXErrorCodeAvailable	449
remainingStack	449
sleep	449
waitFor	449
waitForAllThreads	450
waitForAnyThread	450
Graphical User Interface (GUI) Support	450
anchorBlock	450
appContext	451
appShell	451
initializeGUI	451

isGUIInitialized	452
messageQueue	452
processMsgs	452
terminateGUI	452
Implementation	453
setTopLevelShell	453
setXErrorCode	453
XErrorCode	454
Suspending Threads	454
suspend	454
Inherited Public Functions	454
Protected Functions	455
Constructors	455
ICurrentThread	455
Implementation	455
startedThread	455
Inherited Protected Functions	455
Inherited Protected Data	456
Chapter 65. IEnumHandle	457
Derivation	457
Inherited By	457
Header File	457
Members	457
Public Functions	457
Constructors	457
IEnumHandle	457
Diagnostics	457
asDebugInfo	457
asString	458
asUnsigned	458
Operators	458
operator Value	458
Nested Type Definitions	458
Value	458
Value	458
Chapter 66. IEventData	459
Derivation	459
Inherited By	459
Header File	459
Members	459
Public Functions	460
Constructors	460
IEventData	460
Contents	460
char1	461
char2	461
char3	461
char4	461
highHighByte	461
highLowByte	461
highNumber	461
lowHighByte	461

lowLowByte	462
lowNumber	462
number1	462
number2	462
Conversion	462
asLong	462
asUnsignedLong	462
operator char *	462
operator unsigned long	462
Inherited Public Functions	463
Inherited Protected Data	463
Chapter 67. IEventParameter1	465
Derivation	465
Inherited By	465
Header File	465
Inherited Public Functions	465
Inherited Protected Data	465
Chapter 68. IEventParameter2	467
Derivation	467
Inherited By	467
Header File	467
Inherited Public Functions	467
Inherited Protected Data	467
Chapter 69. IEventResult	469
Derivation	469
Inherited By	469
Header File	469
Inherited Public Functions	469
Inherited Protected Data	469
Chapter 70. IHandle	471
Derivation	471
Header File	471
Members	471
Public Functions	471
Constructors	471
IHandle	472
Diagnostics	472
asDebugInfo	472
asString	472
asUnsigned	472
Operators	472
operator Value	472
Inherited Public Functions	472
Protected Data	472
Value	473
handle	473
Inherited Protected Data	473
Nested Type Definitions	473
Value	473

Chapter 71. IHighEventParameter	475
Derivation	475
Inherited By	475
Header File	475
Inherited Public Functions	475
Inherited Protected Data	475
 Chapter 72. ILowEventParameter	 477
Derivation	477
Inherited By	477
Header File	477
Inherited Public Functions	477
Inherited Protected Data	477
 Chapter 73. INotificationEvent	 479
Derivation	479
Inherited By	479
Header File	479
Members	479
Public Functions	479
Constructors	479
INotificationEvent	479
operator =	480
~INotificationEvent	480
Event Attributes	480
eventData	480
hasNotifierAttrChanged	480
notificationId	480
notifier	481
observerData	481
setEventData	481
setNotifierAttrChanged	481
setObserverData	481
Inherited Public Functions	481
Inherited Protected Data	482
 Chapter 74. INotifier	 483
Derivation	483
Inherited By	483
Header File	483
Members	483
Public Functions	484
Constructors	484
INotifier	484
~INotifier	484
Notification Members	484
disableNotification	484
enableNotification	484
isEnabledForNotification	484
Observer Notification	485
notifyObservers	485
Inherited Public Functions	485
Protected Functions	485
Observer Addition and Removal	485

addObserver	485
observerList	486
removeAllObservers	486
removeObserver	486
Observer Notification	486
notifyObservers	486
Inherited Protected Data	486
Chapter 75. IObserver	487
Derivation	487
Inherited By	487
Header File	487
Members	487
Public Functions	487
Constructors	487
~IObserver	487
Event Dispatching	487
handleNotificationsFor	488
stopHandlingNotificationsFor	488
Inherited Public Functions	488
Protected Functions	488
Constructors	488
IObserver	488
Event Dispatching	489
dispatchNotificationEvent	489
Inherited Protected Data	489
Chapter 76. IObserverList	491
Derivation	491
Inherited By	491
Header File	491
Members	491
Public Functions	491
Constructors	491
IObserverList	491
~IObserverList	492
Observer Addition and Removal	492
add	492
elementAt	492
isEmpty	492
numberOfElements	492
remove	492
removeAll	493
removeAt	493
Observer Notification	493
notifyObservers	493
Inherited Public Functions	493
Inherited Protected Data	493
Nested Classes	494
Chapter 77. IObserverList::Cursor	495
Derivation	495
Inherited By	495
Header File	495

Members	495
Public Functions	495
Constructors	495
Cursor	495
~Cursor	495
Cursor Movement	496
invalidate	496
isValid	496
setToFirst	496
setToLast	496
setToNext	496
setToPrevious	496
Inherited Public Functions	497
Inherited Protected Data	497
 Chapter 78. IPrivateResource	 499
Derivation	499
Inherited By	499
Header File	499
Members	499
Public Functions	500
Constructors	500
IPrivateResource	500
~IPrivateResource	500
Inherited Public Functions	500
Inherited Protected Functions	501
Inherited Protected Data	501
 Chapter 79. IPrivateSemaphoreHandle	 503
Derivation	503
Inherited By	503
Header File	503
Members	503
Public Functions	503
Constructors	503
IPrivateSemaphoreHandle	503
~IPrivateSemaphoreHandle	503
Diagnostics	503
asDebugInfo	504
asString	504
asUnsigned	504
Operators	504
operator =	504
operator Value	504
Nested Type Definitions	504
Value	504
Value	504
Value	504
 Chapter 80. IProcessId	 505
Derivation	505
Inherited By	505
Header File	505
Members	505

Public Functions	505
Constructors	505
IProcessId	505
Diagnostics	505
asDebugInfo	505
asString	506
asUnsigned	506
Operators	506
operator Value	506
Nested Type Definitions	506
Value	506
Value	506
Chapter 81. IRefCounted	507
Derivation	507
Inherited By	507
Header File	507
Members	507
Public Functions	507
Reference Counting	507
addRef	508
removeRef	508
useCount	508
Inherited Public Functions	508
Protected Functions	508
Constructors	508
IRefCounted	508
~IRefCounted	509
Inherited Protected Data	509
Chapter 82. IReference	511
Derivation	511
Inherited By	511
Header File	511
Members	511
Public Functions	512
Constructors	512
IReference	512
operator =	512
~IReference	512
Operators	512
operator *	513
operator ->	513
operator T *	513
Inherited Public Functions	513
Inherited Protected Data	513
Chapter 83. IResource	515
Derivation	515
Inherited By	515
Header File	515
Members	515
Public Functions	515
Constructors	515

IResource	515
~IResource	515
Resource Locking	516
lock	516
unlock	516
Inherited Public Functions	516
Inherited Protected Data	517
Chapter 84. IResourceLock	519
Derivation	519
Inherited By	519
Header File	519
Members	519
Public Functions	519
Constructors	519
IResourceLock	519
~IResourceLock	520
Inherited Public Functions	520
Protected Functions	520
Resource Locking	520
clearLock	520
setLock	521
Inherited Protected Data	521
Chapter 85. ISharedResource	523
Derivation	523
Inherited By	523
Header File	523
Members	523
Public Functions	524
Constructors	524
ISharedResource	524
~ISharedResource	524
Resource Information	525
keyName	525
Inherited Public Functions	525
Inherited Protected Functions	525
Inherited Protected Data	525
Chapter 86. ISharedSemaphoreHandle	527
Derivation	527
Inherited By	527
Header File	527
Members	527
Public Functions	527
Constructors	527
ISharedSemaphoreHandle	527
Diagnostics	527
asDebugInfo	527
asString	528
asUnsigned	528
Operators	528
operator Value	528
Nested Type Definitions	528

Value	528
Value	528
Chapter 87. IStandardNotifier	529
Derivation	529
Inherited By	529
Header File	529
Members	529
Public Functions	529
Constructors	529
IStandardNotifier	530
operator =	530
~IStandardNotifier	530
Notification Members	530
disableNotification	530
enableNotification	530
isEnabledForNotification	531
Observer Notification	531
notifyObservers	531
Inherited Public Functions	531
Protected Functions	531
Observer Addition and Removal	531
addObserver	532
observerList	532
removeAllObservers	532
removeObserver	532
Observer Notification	532
notifyObservers	532
Inherited Protected Functions	533
Public Data	533
Notification Event Descriptions	533
deleteId	533
Inherited Protected Data	533
 Chapter 88. IThread	 535
Derivation	535
Inherited By	535
Header File	535
Members	535
Public Functions	537
Constructors	537
IThread	537
~IThread	539
Diagnostics	539
asDebugInfo	539
asString	540
Graphical User Interface (GUI) Support	540
autoInitGUI	540
defaultAutoInitGUI	540
setAutoInitGUI	540
setDefaultAutoInitGUI	541
stopProcessingMsgs	541
Implementation	541
dialogControls	541

relatedHandlesList	541
setRelatedHandlesList	541
setWindowList	542
windowList	542
Message Queue	542
defaultQueueSize	542
messageQueue	542
queueSize	543
setDefaultQueueSize	543
setQueueSize	543
Stack Size	544
defaultStackSize	544
setDefaultStackSize	544
setStackSize	545
stackSize	545
Starting and Stopping Threads	546
resume	546
start	546
stop	548
suspend	548
Thread Information	548
current	548
currentHandle	549
currentId	549
handle	549
id	549
isStarted	549
setVariable	549
variable	550
Thread Priority	550
adjustPriority	550
priorityClass	550
priorityLevel	551
setPriority	551
Inherited Public Functions	551
Protected Functions	552
Constructors	552
operator =	552
Implementation	552
newStartedThread	552
startedThread	552
Inherited Protected Data	552
Nested Classes	552
Nested Type Definitions	553
(void *)	553
(unsigned long)	553
Chapter 89. IThread::Cursor	555
Derivation	555
Inherited By	555
Header File	555
Members	555
Public Functions	555
Constructors	555

Cursor	556
~Cursor	556
Thread Iteration	556
invalidate	556
isValid	556
setToFirst	556
setToNext	556
threadId	556
Inherited Public Functions	557
Inherited Protected Data	557
Chapter 90. IThreadFn	559
Derivation	559
Inherited By	559
Header File	559
Members	559
Public Functions	559
Constructors	559
IThreadFn	559
~IThreadFn	560
Run Function	560
run	560
Inherited Public Functions	560
Inherited Protected Data	560
Chapter 91. IThreadHandle	561
Derivation	561
Inherited By	561
Header File	561
Members	561
Public Functions	561
Constructors	561
IThreadHandle	561
Diagnostics	561
asDebugInfo	562
asString	562
asUnsigned	562
Public Data	562
Thread Handle Specifics	562
noHandle	562
Nested Type Definitions	562
Value	562
Chapter 92. IThreadId	563
Derivation	563
Inherited By	563
Header File	563
Members	563
Public Functions	563
Constructors	563
IThreadId	563
Diagnostics	563
asDebugInfo	564
asString	564

asUnsigned	564
isValid	564
Operators	564
operator =	564
operator pthread_t	564
Chapter 93. IThreadMemberFn	565
Derivation	565
Inherited By	565
Header File	565
Members	565
Public Functions	565
Constructors	565
IThreadMemberFn	566
~IThreadMemberFn	566
Run Function	566
run	566
Inherited Public Functions	566
Inherited Protected Data	567
Chapter 94. ITime Class	569
Derivation	569
Header File	569
Constructors	569
Public Members	570
asCTIME	570
asSeconds	570
asString	570
hours	570
minutes	570
now	570
operator!=	571
operator+	571
operator+=	571
operator-	571
operator-=	571
operator<	571
operator<<	571
operator<=	571
operator==	571
operator>	571
operator>=	572
seconds	572
Inherited Public Members	572
Protected Members	572
initialize	572
Chapter 95. ITimeStamp	573
Derivation	573
Inherited By	573
Header File	573
Members	573
Public Functions	573
Comparisons	573

operator !=	574
operator <	574
operator <=	574
operator ==	574
operator >	574
operator >=	574
Constructors	575
ITimeStamp	575
Current Date and Time	575
currentTimeStamp	575
Diagnostics	576
asString	576
Manipulations	576
operator +	576
operator +=	576
operator -	576
operator -=	577
Queries	577
asSeconds	577
Type Conversions	577
operator IDate	577
operator ITime	577
Public Data	577
Constants	578
secondsInDay	578
Chapter 96. ITrace Class	579
Derivation	579
Header File	579
Constructors	580
Public Members	581
disableTrace	581
disableWriteLineNumber	581
disableWritePrefix	581
enableTrace	581
enableWriteLineNumber	581
enableWritePrefix	581
isTraceEnabled	581
isWriteLineNumberEnabled	581
isWritePrefixEnabled	582
traceDestination	582
write	582
writeToQueue	582
writeToStandardError	582
writeToStandardOutput	582
Inherited Public Members	582
Protected Members	583
threadId	583
writeFormattedString	583
writeString	583
Enumerations	583
Destination	583

Part 8. Appendixes, Glossary, Bibliography and Index	585
Appendix A. Header Files for Collection Class Library Coding Examples	587
animal.h	587
circle.h	588
curve.h	589
demoelem.h	591
dsur.h	592
graph.h	594
line.h	595
parcel.h	596
planet.h	598
toyword.h	599
transelm.h	600
trmapops.h	601
xebc2asc.h	602
Appendix B. OS/390 C/C++ Class Library Runtime Messages	603
Messages for I/O Stream and Complex Mathematics Class Libraries	603
Messages for Application Support Class Library	603
Messages for Collection Class Library	605
Glossary	609
Bibliography	639
OS/390	639
VS COBOL II Release 4	639
COBOL FOR MVS & VM Release 2	639
COBOL for OS/390 & VM Version 2 Release 1	639
PL/I for MVS & VM Release 1 Modification 1	639
OS PL/I Version 2 Release 3	640
VS FORTRAN Version 2 Release 6	640
CICS/ESA Version 4 Release 1	640
CICS Transaction Server for OS/390 Release 2	640
DB2 Version 3 Release 1	640
DB2 Version 4 Release 1	640
DB2 Version 5 Release 1	640
IMS/ESA Version 4 Release 1	640
IMS/ESA Version 5 Release 1	640
IMS/ESA Version 6 Release 1	641
QMF Version 3 Release 2	641
VSAM	641
Index	643

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 independently 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 may include the names of individuals, companies brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

This publication documents *intended* Programming Interfaces that allow the customer to write OS/390 C/C++ programs.

Any interfaces, including service component interfaces, that are not documented in the OS/390 C/C++ publications are not formal interfaces. You should not build any dependencies on these interfaces, as IBM can change or remove interfaces at any time, without notice.

Any pointers in this publication to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of these Web sites. IBM accepts no responsibility for the content or use of non-IBM Web sites specifically mentioned in this publication or accessed through an IBM Web site that is mentioned in this publication.

Standards

Extracts are reprinted from IEEE Std 1003.1—1990, IEEE Standard Information Technology—Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API) [C language], copyright 1990 by the Institute of Electrical and Electronic Engineers, Inc.

Extracts are reprinted from IEEE P1003.1a Draft 6 July 1991, Draft Revision to Information Technology—Portable Operating System Interface (POSIX), Part 1:

System Application Program Interface (API) [C Language], copyright 1992 by the Institute of Electrical and Electronic Engineers, Inc.

Extracts are reprinted from IEEE Std 1003.2—1992, IEEE Standard Information Technology—Portable Operating System Interface (POSIX)—Part 2: Shells and Utilities, copyright 1990 by the Institute of Electrical and Electronic Engineers, Inc.

Extracts are reprinted from IEEE Std P1003.4a/D6—1992, IEEE Draft Standard Information Technology—Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API)—Amendment 2: Threads Extension [C language], copyright 1990 by the Institute of Electrical and Electronic Engineers, Inc.

Extracts from *ISO/IEC 9899:1990* have been reproduced with the permission of the International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC). The complete standard can be obtained from any ISO or IEC member or from the ISO or IEC Central Offices, Case postale 56, CH - 1211 Geneva 20, Switzerland. Copyright remains ISO and IEC.

Extracts from X/Open Specification, Programming Languages, Issue 4 Release 2, copyright 1988, 1989, February 1992, by the X/Open Company Limited, have been reproduced with the permission of X/Open Company Limited. No further reproduction of this material is permitted without the written notice from the X/Open Company Ltd, UK.

Trademarks

The following terms, which may be denoted by a single asterisk (*), are trademarks of International Business Machines Corporation in the United States or other countries or both:

AD/Cycle	AFP	AIX
AIX/6000	AT	AS/400
BookManager	C Set ++	C/370
C/MVS	C++/MVS	Common User Access
CICS	CICS/ESA	CICSplex
COBOL/370	CUA	CT
DATABASE 2	DB2	DFSMS
DFSMS/MVS	DFSMSdfp	DRDA
ESCON	GDDM	Hiperspace
IBM	IBMLink	IMS
IMS/ESA	MVS/DFP	MVS/ESA
MVS/SP	MVS/XA	Open Class
OpenEdition	Operating System/2	Operating System/400
OS OPEN	OS/2	OS/390
OS/400	PROFS	PS/2
QMF	RACF	RETAIN
S/370	S/390	SAA
SOM	SOMobjects	SP
SQL/DS	System/370	System/390
System Object Model	Systems Application Architecture	VisualAge
VM/ESA	VSE/ESA	VTAM
3090	3890	400

Microsoft, Windows, Windows NT, and the Windows logo are registered trademarks of Microsoft Corporation.

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

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

About This Book

This book provides reference information for the IBM Open Class Library, the comprehensive library of C++ classes that are provided with OS/390 C/C++. IBM Open Class Library consists of the following groups of classes, described individually as “class libraries” in this book:

- The Complex Mathematics Class Library
- The I/O Stream Class Library
- The Collection Class Library
- The Application Support Class Library

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

Who Should Use This Book

This book was written for an audience of skilled C++ programmers, who understand the concept of classes. For individual class libraries you may also need to be familiar with using C++ templates. You should 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

IBM OS/390 C/C++ and Related Publications

This section summarizes the content of the IBM OS/390 C/C++ publications and shows where to find related information in other publications.

Table 1 (Page 1 of 3). OS/390 C/C++ Publications

Book Title and Number	Key Sections/Chapters in the Book
<i>OS/390 C/C++ Programming Guide</i> , SC09-2362	<p>Guidance information for:</p> <ul style="list-style-type: none">• C/C++ input and output• Debugging OS/390 C programs that use input/output• Using linkage specifications in C++• Combining C and assembler• Creating and using DLLs• Using threads in an OS/390 UNIX® application• Reentrancy• Using the decimal data type in C• Handling exceptions, error conditions, and signals• Optimizing code• Optimizing your C/C++ code with Interprocedural Analysis• Network communications under OS/390 OpenEdition• Interprocess communications using OS/390 UNIX services• Structuring a program that uses C++ templates• Using environment variables• Using System Programming C facilities• Library functions for the System Programming C facilities• Using runtime user exits• Using the OS/390 C multitasking facility• Using other IBM products with OS/390 C/C++ (CICS*, CSP, DWS, DB2*, GDDM*, IMS*, ISPF, QMF*)• Direct-to-SOM support under OS/390 C/C++• Internationalization: locales and character sets, code set conversion utilities, mapping variant characters• POSIX character set• Code point mappings• Locales supplied with OS/390 C/C++• Charmap files supplied with OS/390 C/C++• Examples of charmap and locale definition source files• Converting code from code character set IBM-1047• Using built-in functions• Programming considerations for OS/390 UNIX C/C++
<i>OS/390 C/C++ User's Guide</i> , SC09-2361	<p>Guidance information for:</p> <ul style="list-style-type: none">• OS/390 C/C++ examples• Compiler options• Binder options and control statements• Specifying OS/390 Language Environment runtime options• Compiling, IPA Linking, binding, and running OS/390 C/C++ programs• Using precompiled headers• Utilities (Object Library, DLL Rename, CXXFILT, DSECT Conversion, Code Set and Locale, ar and make, BPXBATCH)• Diagnosing problems• Cataloged procedures and REXX EXECs supplied by IBM• Error messages and return codes

Table 1 (Page 2 of 3). OS/390 C/C++ Publications

Book Title and Number	Key Sections/Chapters in the Book
<i>OS/390 C/C++ Language Reference</i> , SC09-2360	Reference information for: <ul style="list-style-type: none"> • The C and C++ Languages • Lexical elements of OS/390 C and OS/390 C++ • Declarations, expressions and operators • Implicit type conversions • Functions and statements • Preprocessor directives • C++ classes, class members, and friends • C++ overloading, special member functions, and inheritance • C++ templates and exception handling • OS/390 C and OS/390 C++ compatibility
<i>OS/390 C/C++ Run-Time Library Reference</i> , SC28-1663	Reference information for: <ul style="list-style-type: none"> • C header files • C Library functions
<i>OS/390 C Curses</i> , SC28-1907	Reference information for: <ul style="list-style-type: none"> • Curses concepts • Key data types • General rules for characters, renditions, and window properties • General rules of operations and operating modes • Use of macros • Restrictions on block-mode terminals • Curses functional interface • Contents of headers • The terminfo database
<i>OS/390 C/C++ Compiler and Run-Time Migration Guide</i> , SC09-2359	Guidance and reference information for: <ul style="list-style-type: none"> • Common migration questions • Application executable program compatibility • Source program compatibility • Input and output operations compatibility • Class library migration considerations • Changes between releases of OS/390 • C/370* V1 to V2 compiler changes • Other migration considerations
<i>OS/390 C/C++ Reference Summary</i> , SX09-1313	Summary tables for: <ul style="list-style-type: none"> • Character set, trigraphs, digraphs, and keywords • Escape sequences, storage classes • Predefined and derived types, type qualifiers • Operator precedence, redirection symbols • fprintf format, type characters, and flag characters • fscanf format and type characters • __amrc structure • Hardware exceptions and signals • Compiler return codes • Compiler options • #pragma directives • Library functions • Utilities

Table 1 (Page 3 of 3). OS/390 C/C++ Publications

Book Title and Number	Key Sections/Chapters in the Book
<i>OS/390 C/C++ IBM Open Class Library User's Guide</i> , SC09-2363	<p>Guidance information for:</p> <ul style="list-style-type: none"> • Using the Complex Mathematics Class Library: Review of complex numbers, header files, constructing complex objects, mathematical operators for complex, friend functions for complex, handling complex mathematical errors • Using the I/O Stream Class Library: Introduction, getting started, advanced topics, and manipulators • Using the Collection Class Library: Overview, instantiating and using, Element and Key functions, tailoring collection implementation, polymorphic use of collections, support for notifications, exception handling, tutorials, problem solving, compatibility with previous releases, thread safety • Using the Application Support Class Library: Introduction, String classes, Exception and Trace classes, Date and Time classes, controlling threads and protecting data, the IBM Open Class* notification framework, Binary Coded Decimal classes
<i>OS/390 C/C++ IBM Open Class Library Reference</i> , SC09-2364	<p>Reference information for:</p> <ul style="list-style-type: none"> • Complex Mathematics Class Library • I/O Stream Class Library • Collection Class Library • Application Support Class Library
<i>OS/390 C/C++ SOM-Enabled Class Library User's Guide and Reference</i> , SC09-2366	<p>Guidance and reference information for:</p> <ul style="list-style-type: none"> • C++ SOM (RRBC-enabled) versions of Collection and Application Support Class Libraries • Cross-language SOM version of the Collection Class Library
<i>Debug Tool User's Guide and Reference</i> , SC09-2137	<p>Guidance and reference information for:</p> <ul style="list-style-type: none"> • Preparing to debug programs • Debugging programs • Using Debug Tool in different environments • Language-specific information • Debug Tool reference
APAR and BOOKS files (Shipped with Program materials)	<p>Partitioned data set CBC.SCBCDOC on the product tape contains the members, APAR and BOOKS, which provide additional information for using the IBM OS/390 C/C++ licensed program, including:</p> <ul style="list-style-type: none"> • Isolating reportable problems • Keywords • Preparing an Authorized Program Analysis Report (APAR) • Problem identification worksheet • Maintenance on OS/390 • Late changes to OS/390 C/C++ publications

Note: For complete and detailed information on linking and running with OS/390 Language Environment and using the OS/390 Language Environment runtime options, refer to the *OS/390 Language Environment Programming Guide*, SC28-1939. For complete and detailed information on using interlanguage calls, refer to *OS/390 Language Environment Writing Interlanguage Applications*, SC28-1943.

The following table lists the OS/390 C/C++ and related publications that you are most likely to need. Publications are grouped according to the tasks they describe.

Table 2 (Page 1 of 4). Publications by Task

Tasks	Books
Planning, preparing, and migrating to OS/390 C/C++	<p><i>OS/390 C/C++ Compiler and Run-Time Migration Guide</i>, SC09-2359</p> <p><i>OS/390 Language Environment Concepts Guide</i>, GC28-1945</p> <p><i>OS/390 Language Environment Customization</i>, SC28-1941</p> <p><i>OS/390 Planning for Installation</i>, GC28-1726</p> <p>OS/390 Task Atlas, available on the OS/390 Library page on the World Wide Web (http://www.s390.ibm.com/os390/bkserv)</p>
Installing	<p>OS/390 Program Directory</p> <p><i>OS/390 Planning for Installation</i>, GC28-1726</p> <p><i>OS/390 Language Environment Customization</i>, SC28-1941</p>
Coding programs	<p><i>OS/390 C/C++ Run-Time Library Reference</i>, SC28-1663</p> <p><i>OS/390 C/C++ Language Reference</i>, SC09-2360</p> <p><i>OS/390 C/C++ Reference Summary</i>, SX09-1313</p> <p><i>OS/390 C/C++ Programming Guide</i>, SC09-2362</p> <p><i>OS/390 Language Environment Concepts Guide</i>, GC28-1945</p> <p><i>OS/390 Language Environment Programming Guide</i>, SC28-1939</p> <p><i>OS/390 Language Environment Programming Reference</i>, SC28-1940</p> <p><i>OS/390 C/C++ IBM Open Class Library User's Guide</i>, SC09-2363</p> <p><i>OS/390 C/C++ IBM Open Class Library Reference</i>, SC09-2364</p> <p><i>OS/390 C/C++ SOM-Enabled Class Library User's Guide and Reference</i>, SC09-2366</p>

Table 2 (Page 2 of 4). Publications by Task

Tasks	Books
Coding and binding programs with interlanguage calls	<p><i>OS/390 C/C++ Programming Guide</i>, SC09-2362</p> <p><i>OS/390 C/C++ Language Reference</i>, SC09-2360</p> <p><i>OS/390 Language Environment Programming Guide</i>, SC28-1939</p> <p><i>OS/390 Language Environment Writing Interlanguage Applications</i>, SC28-1943</p> <p><i>DFSMS/MVS Program Management</i>, SC28-1943</p>
Compiling, binding, and running programs	<p><i>OS/390 C/C++ User's Guide</i>, SC09-2361</p> <p><i>OS/390 Language Environment Programming Guide</i>, SC28-1939</p> <p><i>OS/390 Language Environment Debugging Guide and Run-Time Messages</i>, SC28-1942</p> <p><i>DFSMS/MVS Program Management</i>, SC26-4916</p> <p>OS/390 Messages Database, available from the OS/390 Library page in the World Wide Web (http://www.s390.ibm.com/os390/bkserv)</p>
Compiling and binding applications in the OS/390 OpenEdition environment	<p><i>OS/390 C/C++ User's Guide</i>, SC09-2361</p> <p><i>OS/390 UNIX System Services User's Guide</i>, SC28-1891</p> <p><i>OS/390 UNIX System Services Command Reference</i>, SC28-1892</p> <p><i>DFSMS/MVS Program Management</i>, SC26-4916</p>
Compiling and binding SOM applications with OS/390 SOMobjects*	<p><i>OS/390 SOMobjects Programmer's Guide</i>, GC28-1859</p> <p><i>OS/390 C/C++ Programming Guide</i>, SC09-2362</p> <p><i>OS/390 C/C++ User's Guide</i>, SC09-2361</p>

Table 2 (Page 3 of 4). Publications by Task

Tasks	Books
Debugging programs	<p>README file</p> <p><i>Debug Tool User's Guide and Reference</i>, SC09-2137</p> <p><i>OS/390 C/C++ User's Guide</i>, SC09-2361</p> <p><i>OS/390 C/C++ Programming Guide</i>, SC09-2362</p> <p><i>OS/390 Language Environment Programming Guide</i>, SC28-1939</p> <p><i>OS/390 Language Environment Debugging Guide and Run-Time Messages</i>, SC28-1942</p> <p><i>OS/390 UNIX System Services Messages and Codes</i>, SC28-1908</p> <p><i>OS/390 UNIX System Services User's Guide</i>, SC28-1891</p> <p><i>OS/390 UNIX System Services Command Reference</i>, SC28-1892</p> <p><i>OS/390 UNIX System Services Programming Tools</i>, SC28-1904</p>
Using shells and utilities in the OS/390 OpenEdition environment	<p><i>OS/390 C/C++ User's Guide</i>, SC09-2361</p> <p><i>OS/390 UNIX System Services Command Reference</i>, SC28-1892</p> <p><i>OS/390 UNIX System Services Messages and Codes</i>, SC28-1908</p>
Using sockets library functions in the OS/390 OpenEdition environment	<p><i>OS/390 C/C++ Run-Time Library Reference</i>, SC28-1663</p>
Porting a UNIX Application to OS/390	<p><i>OS/390 UNIX System Services Porting Guide</i></p> <p>This guide contains useful information about supported header files and C functions, sockets in an OS/390 UNIX environment, process management, compiler optimization tips, and suggestions for improving the application's performance after it has been ported. The <i>Porting Guide</i> is available as a PDF file which you can download, or as web pages which you can browse, at the following URL:</p> <p>http://www.s390.ibm.com/unix/bpxa1por.html</p>
Performing diagnosis and submitting an Authorized Program Analysis Report (APAR)	<p><i>OS/390 C/C++ User's Guide</i>, SC09-2361</p> <p>CBC.SCBCDOC(APAR) on OS/390 C/C++ product tape</p>
Quick reference	<p><i>OS/390 C/C++ Reference Summary</i>, SX09-1313</p>

Table 2 (Page 4 of 4). Publications by Task

Tasks	Books
Multimedia Tutorial	For a new way of learning C++ programming, you can order the CD-ROM <i>Experience C++: A Multimedia Tutorial</i> , SK2T-1158. This tutorial runs in DOS.

Note: For information on using the prelinker, see the appendix on prelinking and linking OS/390 C/C++ programs in the *OS/390 C/C++ User's Guide*. As of Release 4, this appendix contains information that was previously in the chapter on prelinking and linking OS/390 C/C++ programs in the *OS/390 C/C++ User's Guide*. It also contains prelinker information that was previously in the *OS/390 C/C++ Programming Guide*.

Hardcopy Books

You can purchase OS/390 C/C++ books one at a time, or in a set. The following OS/390 C/C++ books are available in hardcopy:

- *OS/390 C/C++ Run-Time Library Reference*, SC28-1663
- *OS/390 C/C++ User's Guide*, SC09-2361
- *OS/390 C/C++ Programming Guide*, SC09-2362
- *OS/390 C/C++ Reference Summary*, SX09-1313
- *OS/390 C/C++ IBM Open Class Library User's Guide*, SC09-2363
- *OS/390 C Curses*, SC28-1907
- *OS/390 C/C++ Compiler and Run-Time Migration Guide*, SC09-2359
- *Debug Tool User's Guide and Reference*, SC09-2137

These books can be purchased singly or as part of a set. The *OS/390 C/C++ Compiler and Run-Time Migration Guide*, SC09-2359 is provided at no charge. The remaining books are included in feature code 8009.

Softcopy Books

All of the OS/390 C/C++ publications (except for the *OS/390 C/C++ Reference Summary*) are available in softcopy book format. The books are available on a tape accompanying the OS/390 product, and also on a CD-ROM called the *IBM Online Library Omnibus Edition: OS/390 Collection*, SK2T-6700.

To read the softcopy books, the BookManager* Read (Program 5684-062, 5695-046) licensed program must be available on your operating system. BookManager Read provides access to online information as an alternative to hard copy documents. You can read, search, make notes, and select sections of text to print.

Also available are BookManager Read/DOS (Program 73F6-022) for the DOS operating system, and BookManager Read/2 (Program 73F6-023) for the OS/2 operating system. With these products, you can download online books to your workstation and read them.

With BookManager Read installed on your system, you can enter the command BOOKMGR to start BookManager and display a list of books available to you. If you know the name of the book that you want to view, you can use the OPEN command to open the book directly.

Note: If your workstation does not have graphics capability, BookManager Read cannot correctly display some characters, such as arrows and brackets.

You can also browse the books on the World Wide Web, through "The Library" link on the OS/390 home page. The URL for this page is:

<http://www.s390.ibm.com/os390/index.html>

Softcopy Examples

Most of the larger examples in the following books are available in machine-readable form:

- *OS/390 C/C++ Language Reference*, SC09-2360
- *OS/390 C/C++ User's Guide*, SC09-2361
- *OS/390 C/C++ Programming Guide*, SC09-2362
- *OS/390 C/C++ IBM Open Class Library User's Guide*, SC09-2363
- *OS/390 C/C++ IBM Open Class Library Reference*, SC09-2364
- *OS/390 C/C++ SOM-Enabled Class Library User's Guide and Reference*, SC09-2366

In the following books, a label on an example indicates that the example is distributed in softcopy. The label is the name of a member in the data sets CBC.SCBCSAM or CBC.SCLBSAM. The labels have the form CBCxyyy or CLBxyyy, where x refers to a publication:

- R and X refer to the *OS/390 C/C++ Language Reference*, SC09-2360
- G refers to the *OS/390 C/C++ Programming Guide*, SC09-2362
- U refers to the *OS/390 C/C++ User's Guide*, SC09-2361
- A refers to the *OS/390 C/C++ IBM Open Class Library User's Guide*, SC09-2363

Examples labelled as CBCxyyy appear in the *OS/390 C/C++ Language Reference*, the *OS/390 C/C++ Programming Guide*, and the *OS/390 C/C++ User's Guide*. Examples labelled as CLBxyyy appear in the *OS/390 C/C++ IBM Open Class Library User's Guide*.

An exception applies to the example names for the Collection Class Library, which do not follow a naming convention. These examples are in this book and in the *OS/390 C/C++ SOM-Enabled Class Library User's Guide and Reference*.

OS/390 C/C++ on the World Wide Web

Additional information on OS/390 C/C++ is available on the World Wide Web. The URL for the OS/390 C/C++ home page is:

<http://www.software.ibm.com/ad/c390/>

This page contains late-breaking information about the OS/390 C/C++ product, including the compiler, the class libraries, and utilities. It also contains a tutorial on the source level interactive debugger. There are links to other useful information,

such as the OS/390 C/C++ information library and the libraries of other OS/390 elements that are available on the Web. The OS/390 C/C++ home page also contains information on active Beta programs, code samples that you can download, the C/370 product newsletters, and links to other related Web sites.

C/C++ News...

IBM also publishes the *C/370 Compiler Newsletter*. This free newsletter keeps subscribers up to date on the latest product releases, provides coding hints and tips, questions and answers, and news about C/370 products and IBM OS/390 C/C++.

To take advantage of this free publication, send your name, full mailing address, and phone number, in one of these ways:

- Send a message electronically to the following network ID :

- Internet: `inetc370@vnet.ibm.com`
- IBMMAIL: `ibmmail(caibmrzx)`

- Mail your request to:

EDITOR, C/370 Compiler Newsletter
IBM Canada Ltd. Laboratory
9/604/895/TOR
895 Don Mills Road
NORTH YORK ONTARIO CANADA M3C 1W3

About IBM OS/390 C/C++

The C/C++ feature of the IBM OS/390 licensed program provides support for C and C++ application development on the OS/390 platform. The C/C++ feature is based on the C/C++ for MVS/ESA* product.

IBM OS/390 C/C++ includes:

- A C compiler (referred to as the OS/390 C compiler)
- A C++ compiler (referred to as the OS/390 C++ compiler)
- A set of C++ class libraries
- Application Support Class and Collection Class Library source
- A mainframe interactive Debug Tool (optional)
- A set of utilities for C/C++ application development

IBM offers the C language on other platforms, such as the AIX*, IBM Operating System/2* (OS/2*), IBM Operating System/400* Version 3 (OS/400*), Sun Solaris, VM/ESA*, VSE/ESA*, and Windows® operating systems. The AIX, OS/2, OS/400, Sun Solaris, and Windows operating systems also offer the C++ language.

Changes for Version 2 Release 6

OS/390 C/C++ has made the following changes for this release:

- Added support for the Institute of Electrical and Electronics Engineers (IEEE) binary floating-point data type, in conformance with the IEEE 754 standard, as applicable to the S/390* environment. For details on the OS/390 C/C++ support, see the description of the FLOAT option in the *OS/390 C/C++ User's Guide*. In addition, two related sub-options have been introduced, ARCH(3) and TUNE(3). The two sub-options support the new G5 processor architecture, and IEEE binary floating-point data. Refer to the ARCHITECTURE and TUNE compiler options in the *OS/390 C/C++ User's Guide* for details.

Complete IEEE binary floating-point support for OS/390 and its elements requires that you apply small programming enhancements (SPEs) to OS/390 V2R6.0, and to specific releases of some software. These SPEs are delivered as program temporary fixes (PTFs). Consult your System Programmer to ensure that the SPE PTFs you require for IEEE binary floating-point support, as documented in the *OS/390 Planning for Installation* publication, are applied to your system. The *OS/390 Planning for Installation* publication documents the complete software requirements for IEEE binary floating-point support on OS/390.

- Improved the performance of the Binary Coded Decimal (BCD) class library, and its compatibility with the decimal data type in C, and other S/390 languages. For details, see *Using the C++ Decimal Data Type* in the *OS/390 C/C++ Programming Guide*.
- Added support for the long long integer data type. For more details, see the sections on integer declarations in the *OS/390 C/C++ Language Reference*. The run-time library, including functions such as printf() and scanf(), does not support the long long data type at this time.
- Added a new compiler option, PORT, that enables you to increase the syntax checking for the #pragma pack directive in your code. This option is helpful

when porting code that contains `#pragma pack` directives or packed data from other platforms. For more information on the `PORT` option, see the *OS/390 C/C++ User's Guide*.

- Added a new compiler option, `FASTTEMPINC`, that enables you to improve your compilation time for C++ class templates if you use a large number of recursive templates in an application. For more information on the `FASTTEMPINC` option, see the *OS/390 C/C++ User's Guide*.
- Retroactive to OS/390 Version 1 Release 3, the IBM Open Class Library is licensed with the base operating system. This enables applications to use this library at run time without having to license the OS/390 C/C++ compiler feature(s) or to use the DLL Rename Utility.
- The level of optimization you get when you specify the `OPT(1)`, or `OPT`, compiler option is the same as when you specify `OPT(2)`. For more information on the `OPTIMIZATION` option see the *OS/390 C/C++ User's Guide*.
- The OS/390 C++ class library header files are now distributed in the hierarchical file system (HFS) in directory `/usr/lpp/ioclib/include`.
- As part of the name change of *OpenEdition** to *OS/390 UNIX System Services*, occurrences of *OpenEdition* have been changed to *OS/390 UNIX System Services* or its abbreviated name, *OS/390 UNIX*, throughout the OS/390 C/C++ information library. *OpenEdition* may continue to appear in messages, panel text, and other code locations.

The C/C++ Compilers

The following sections describe the C and C++ languages and the OS/390 C/C++ compilers.

The C Language

The C language is a general purpose, versatile, and functional programming language, which allows a programmer to create applications quickly and easily. C provides high-level control statements and data types as do other structured programming languages. It also provides many of the benefits of a low-level language.

The C++ Language

The C++ language is based on the C language, but incorporates support for object-oriented concepts. For a detailed description of the differences between OS/390 C++ and OS/390 C, refer to the *OS/390 C/C++ Language Reference*.

The C++ language introduces classes, which are user-defined data types that may contain data definitions and function definitions. You can use classes from established class libraries, develop your own classes, or derive new classes from existing classes by adding data descriptions and functions. New classes can inherit properties from one or more classes. Not only do classes describe the data types and functions available, but they can also hide (encapsulate) the implementation details from user programs. An object is an instance of a class.

The C++ language also provides templates and other features that include access control to data and functions, and better type checking and exception handling. It also supports polymorphism and the overloading of operators.

Common Features of the OS/390 C and C++ Compilers

The C or C++ compilers offer many features to help your work:

- Optimization support.
 - Algorithms to take advantage of S/390 architecture to get better optimization for speed and use of computer resources through the OPTIMIZE and IPA compile-time options.
 - The OPTIMIZE compile-time option to instruct the compiler to optimize the machine instructions it generates, to produce faster-running object code, thereby optimizing application performance at run time.
 - Interprocedural Analysis (IPA), to perform optimizations across compilation units, thereby optimizing application performance at run time.
 - The precompiled header facility, to save information from one compilation unit for use in another or to reuse information when re-compiling the source compilation unit, thereby improving performance at compile time.

- DLLs (dynamic link libraries) to reduce application size, and dynamically link to exported variables and functions at run time.

IBM OS/390 C/C++ provides support for generating DLLs in a way similar to the way OS/2 generates DLLs. DLLs allow a function reference or a variable reference in one executable to use a definition located in another executable at run time. You can use both load-on-reference and load-on-demand DLLs. When your program calls a DLL function, or references a DLL, IBM OS/390 C/C++ provides a load-on-reference DLL. Your application code explicitly controls load-on-demand DLLs at the source level.

You can use DLLs to split applications into smaller modules and improve system memory usage. DLLs also offer more flexibility for building, packaging, and redistributing applications.

- Full program reentrancy.

With reentrancy, many users can simultaneously run a program. A reentrant program uses less storage if it is stored in the LPA (link pack area) or ELPA (extended link pack area) and simultaneously run by multiple users. It also reduces processor I/O when the program starts up, and improves program performance by reducing the transfer of data to auxiliary storage. OS/390 C programmers can design programs that are naturally reentrant. For those programs that are not naturally reentrant, C programmers can use constructed reentrancy. To do this, compile programs with the RENT option and use the program management binder supplied with OS/390, or the OS/390 Language Environment Prelinker (prelinker) and program management binder. The OS/390 C++ compiler always ensures that C++ programs are reentrant.

- Locale-based internationalization support derived from the IEEE POSIX 1003.2-1992 standard. Also derived from the X/Open CAE Specification, System Interface Definitions, Issue 4 and Issue 4 Version 2. This allows programmers to use locales to specify language/country characteristics for their applications.
- The ability to call and be called by other languages such as assembler, COBOL, PL/1, and Fortran, to enable programmers to integrate OS/390 C/C++ code with existing applications.
- Exploitation of OS/390 and OS/390 UNIX technology.

OS/390 UNIX is an IBM implementation of the open operating system environment, as defined in the XPG4 and POSIX standards.

- When used with OS/390 UNIX and OS/390 Language Environment, support for the following standards at the system level:
 - A subset of the extended multibyte and wide character functions as defined by the Programming Language C Amendment 1. This is ISO/IEC 9899:1990/Amendment 1:1994(E)
 - ISO/IEC 9945-1:1990(E)/IEEE POSIX 1003.1-1990
 - A subset of IEEE POSIX 1003.1a, Draft 6, July 1991
 - IEEE Portable Operating System Interface (POSIX) Part 2, P1003.2
 - A subset of IEEE POSIX 1003.4a, Draft 6, February 1992 (the IEEE POSIX committee has renumbered POSIX.4a to POSIX.1c)
 - X/Open CAE Specification, System Interfaces and Headers, Issue 4 Version 2
 - A subset of IEEE 754-1985 (R1990) IEEE Standard for Binary Floating-Point Arithmetic (ANSI), as applicable to the S/390 environment.
 - X/Open CAE Specification, Network Services, Issue 4
- Year 2000 support.

OS/390 C Compiler Specific Features

In addition to the features common to OS/390 C/C++, the OS/390 C compiler provides you with the following capabilities:

- The ability to write portable code that conforms to the following standards:
 - All elements of the ISO standard ISO/IEC 9899:1990 (E)
 - ANSI/ISO 9899:1990[1992] (formerly ANSI X3.159-1989 C)
 - X/Open Specification Programming Language Issue 3, Common Usage C
 - FIPS-160
- System programming capabilities, which allow you to use OS/390 C in place of assembler
- Additional optimization capabilities through the `INLINE` compile-time option
- Extensions of the standard definitions of the C language to provide programmers with support for the OS/390 environment, such as fixed-point (packed) decimal data support

Features That Are Specific to the OS/390 C++ Compiler

In addition to the features common to OS/390 C/C++, the OS/390 C++ compiler provides you with the following:

- An implementation based on the definition of the language that is contained in the Draft Proposal International Standard for Information Systems—Programming Language C++ (X3J16/92-00091). The OS/390 C++ compiler also conforms to a subset of the C++ ANSI/ISO (Draft) Standard (X3J16/93-0062).
- System Object Model (SOM) support, through the SOM Interface Definition Language (IDL) compiler available with OS/390 SOMobjects. You can use the IDL compiler and associated emitters to create language-specific bindings that

define the interface to a SOM object. This enables OS/390 C++ programs to share SOM objects with other languages. In addition, SOM enables release-to-release binary compatibility.

With Direct-to-SOM (DTS) support in the OS/390 C++ compiler, you can generate SOM objects directly from C++ code. You do not need to create and process the IDL first. You can write virtually the same code you do when creating C++ objects.

Note: The OS/390 C++ compiler no longer supports IDL generation through the IDL compile-time option. This option instructed the compiler to generate IDL. Mixed-language or distributed object applications used IDL. If you need IDL for your applications, you should now code it yourself instead of generating it through the IDL compile option.

- C++ template support and exception handling consistent with VisualAge* C++ product implementations.

Utilities

The OS/390 C/C++ compilers provide the following utilities:

- The Object Library Utility to update partitioned data set (PDS) libraries of object modules and Interprocedural Analysis (IPA) object modules
- The DLL Rename Utility to make selected DLLs a unique component of the applications with which they are packaged
- The CXXFILT Utility to map OS/390 C++ mangled names to the original source
- The localedef Utility to read the locale definition file and produce a locale object that the locale-specific library functions can use
- The DSECT Conversion Utility to convert descriptive assembler DSECTs into OS/390 C/C++ data structures
- The C/C++ Model Tool to provide online help for C/C++ #pragma directives and runtime library functions. These functions are other than the C Curses functions, and are at the level that is supplied in OS/390 Release 2

Class Libraries

IBM OS/390 C/C++ provides a base set of class libraries, called C/C++ IBM Open Class, which is consistent with that available in other members of the VisualAge C++ product family. These class libraries are:

- The I/O Stream Class Library

The I/O Stream Class Library lets you perform input and output (I/O) operations independent of physical I/O devices or data types that are used. You can code sophisticated I/O statements easily and clearly, and define input and output for your own data types. You can improve the maintainability of programs that use input and output by using the I/O Stream Class Library.

- The Complex Mathematics Class Library

The Complex Mathematics Class Library lets you manipulate and perform standard arithmetic on complex numbers. Scientific and technical fields use complex numbers.

- The Application Support Class Library

The Application Support Class Library provides the basic abstractions that are needed during the creation of most C++ applications, including String, Date, and Time.

The Application Support Class library is available in a C++ SOM version as well as the regular C++ native version.

- The Collection Class Library

The Collection Class Library implements a wide variety of classical data structures such as stack, tree, list, hash table, and so on. Most programs use collections. You can develop programs without having to define every collection. Programmers can start programming by using a high level of abstraction, and later replace an abstract data type with the appropriate concrete implementation. Each abstract data type has a common interface for all of its implementations. The Collection Class Library provides programmers with a consistent set of building blocks from which they can derive application objects. The library design exploits features of the C++ language such as exception handling and template support.

The Collection Class Library is available in a C++ SOM and a cross-language SOM version, as well as the regular C++ native version.

All of the libraries that are described above are thread-safe, except the cross-language SOM version of the Collection Class Library.

All of the libraries that are described above are available in both static and DLL formats. OS/390 C/C++ packages the Application Support Class and Collection Class libraries together in a single DLL. For compatibility, separate side-decks are available for the Application Support Class and Collection Class libraries, in addition to the side-deck available for the combined library.

Note: Retroactive to OS/390 Version 1 Release 3, the IBM Open Class Library is licensed with the base operating system. This enables applications to use this library at run time without having to license the OS/390 C/C++ compiler feature(s) or to use the DLL Rename Utility.

Class Library Source

The Class Library Source consists of the following:

- Application Support Class Library source code
- Collection Class Library source code (C++ native and C++ SOM only)
- Instructions for building the Application Support Class and Collection Class Libraries in C++ native (static and DLL) versions
- Instructions for building the Application Support Class and Collection Class Libraries in C++ SOM (static and DLL) versions
- Class Library Language Environment message file source
- Instructions for building the Class Library Language Environment message files

The Debug Tool

IBM OS/390 C/C++ supports program development by using a mainframe interactive Debug Tool. This optionally available tool allows you to debug applications in their native host environment, such as CICS/ESA, IMS/ESA*, DB2, and so on. The Debug Tool provides the following support and function:

- Step mode
- Breakpoints
- Monitor
- Frequency analysis
- Dynamic patching

You can record the debug session in a log file, and replay the session. You can also use the Debug Tool to help capture test cases for future program validation or to further isolate a problem within an application.

You can specify either data sets or hierarchical file system (HFS) files as source files.

OS/390 Language Environment

IBM OS/390 C/C++ exploits the C/C++ runtime environment and library of runtime services available with OS/390 Language Environment (formerly Language Environment for MVS & VM, Language Environment/370 and LE/370).

OS/390 Language Environment consists of four language-specific runtime libraries, and Base Routines and Common Services; see Figure 1. OS/390 Language Environment establishes a common runtime environment and common runtime services for language products, user programs, and other products.

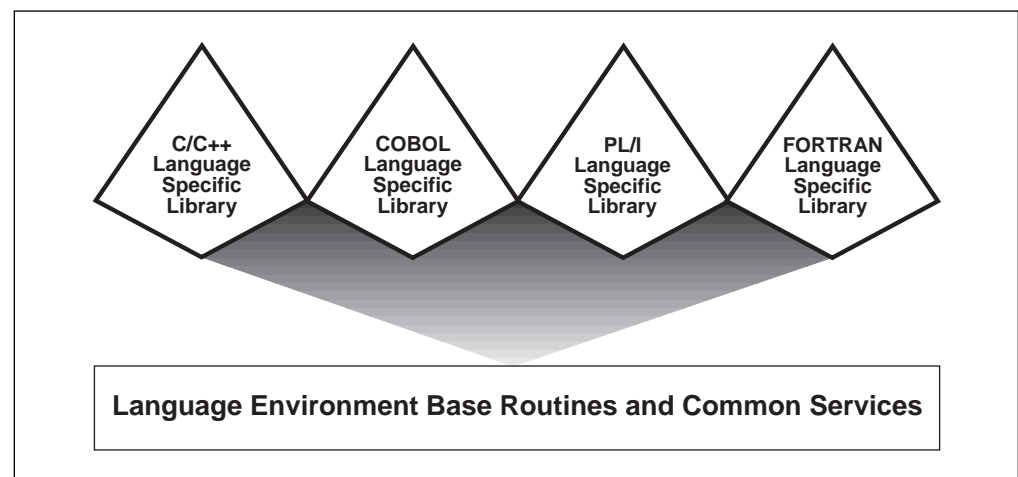


Figure 1. Libraries in OS/390 Language Environment

The common execution environment is composed of data items and services that are included in library routines available to an application that runs in the environment. The OS/390 Language Environment provides a variety of services:

- Services that satisfy basic requirements common to most applications. These include support for the initialization and termination of applications, allocation of storage, interlanguage communication (ILC), and condition handling.

- Extended services that are often needed by applications. OS/390 C/C++ contains these functions within a library of callable routines, and include interfaces to operating system functions and a variety of other commonly used functions.
- Runtime options that help in the execution, performance, and diagnosis of your application.
- Access to operating system services; OS/390 UNIX services are available to an application programmer or program through the OS/390 C/C++ language bindings.
- Access to language-specific library routines, such as the OS/390 C/C++ library functions.

The Program Management Binder

The binder provided with OS/390 combines the object modules, load modules, and program objects comprising an OS/390 application. It produces a single output program object or load module that you can load for execution. The binder supports all C and C++ code, provided that you store the output program in a PDSE (Partitioned Data Set Extended) member or an HFS file.

If you cannot use a PDSE member or HFS file, and your program contains C++ code, or C code that is compiled with any of the RENT, LONGNAME, DLL or IPA compile-time options, you must use the prelinker.

Using the binder without using the prelinker has the following advantages:

- Faster rebinds when recompiling and rebinding a few of your source files
- Rebinding at the single compile unit level of granularity (except when you use the IPA compile-time option)
- Input of object modules, load modules, and program objects
- Improved long name support:
 - Long names do not get converted into prelinker generated names
 - Long names appear in the binder maps, enabling full cross-referencing
 - Variables do not disappear after prelink
 - Fewer steps in the process of producing your executable program

The prelinker provided with OS/390 Language Environment combines the object modules comprising an OS/390 C/C++ application and produces a single object module. You can link-edit the object module into a load module (which is stored in a PDS), or bind it into a load module or a program object stored in a PDS, or a PDSE or HFS file.

OS/390 UNIX System Services (OS/390 UNIX)

OS/390 UNIX provides capabilities under OS/390 to make it easier to implement or port applications in an open, distributed environment. OS/390 UNIX Services are available to OS/390 C/C++ application programs through the C/C++ language bindings available with OS/390 Language Environment.

Together, the OS/390 UNIX Services, OS/390 Language Environment, and OS/390 C/C++ compilers provide an application programming interface that supports industry standards.

OS/390 UNIX provides support for both existing OS/390 applications and new OS/390 UNIX applications:

- C programming language support as defined by ISO/ANSI C
- C++ programming language support
- C language bindings as defined in the IEEE 1003.1 and 1003.2 standards; subsets of the draft 1003.1a and 1003.4a standards; X/Open CAE Specification: System Interfaces and Headers, Issue 4, Version 2, which provides standard interfaces for better source code portability with other conforming systems; and X/Open CAE Specification, Network Services, Issue 4, which defines the X/Open UNIX descriptions of sockets and X/Open Transport Interface (XTI)
- OS/390 UNIX Extensions that provide OS/390-specific support beyond the defined standards
- The OS/390 UNIX Shell and Utilities feature, which provides:
 - A shell, based on the Korn Shell and compatible with the Bourne Shell
 - Tools and utilities that conform to the *X/Open Single UNIX Specification*, also known as *X/Open Portability Guide (XPG) Version 4, Issue 2*, and provide OS/390 support. The following utilities are included:

ar Creates and maintains library archives

BPXBATCH Allows you to submit batch jobs that run shell commands, scripts, or OS/390 C/C++ executable files in HFS files from a shell session

c89 Compiles, assembles, and binds OS/390 UNIX C applications

gencat Merges the message text source files Messagefile (usually *.msg) into a formatted message Catalogfile (usually *.cat)

lex Automatically writes large parts of a lexical analyzer based on a description that is supplied by the programmer

make Helps you manage projects containing a set of interdependent files, such as a program with many OS/390 C/C++ source and object files, keeping all such files up to date with one another

yacc Allows you to write compilers and other programs that parse input according to strict grammar rules

- Support for other utilities such as:

c++ Compiles, assembles, and binds OS/390 UNIX C++ applications

mkcatdefs Preprocesses a message source file for input to the gencat utility

runcat Invokes mkcatdefs and pipes the message catalog source data (the output from mkcatdefs) to gencat

dspcat	Displays all or part of a message catalog
dspmsg	Displays a selected message from a message catalog

- The OS/390 UNIX Debugger feature, which provides the dbx interactive symbolic debugger for OS/390 UNIX applications
- OS/390 UNIX, which provides access to a hierarchical file system (HFS), with support for the POSIX.1 and XPG4 standards
- OS/390 C/C++ I/O routines, which support using HFS files, standard OS/390 data sets, or a mixture of both
- Application threads (with support for a subset of POSIX.4a)
- Support for OS/390 C/C++ DLLs

OS/390 UNIX offers program portability across multivendor operating systems, with support for POSIX.1, POSIX.1a (draft 6), POSIX.2, POSIX.4a (draft 6), and XPG4.2.

To application developers who have worked with other UNIX environments, the OS/390 UNIX Shell and Utilities are a familiar environment for C/C++ application development. If you are familiar with existing MVS development environments, you may find that the OS/390 UNIX environment can enhance your productivity. Refer to the *OS/390 UNIX System Services User's Guide* for more information on the Shell and Utilities.

OS/390 C/C++ Applications with OS/390 UNIX C/C++ Functions

Most OS/390 UNIX C functions are available at all times. However, to use some OS/390 UNIX C functions, you must run an OS/390 C/C++ program on a system where the OS/390 UNIX kernel is available and active. In some situations, you must also specify the `POSIX(ON)` runtime option. This is required for the POSIX.4a threading functions, and the system and signal handling functions where the behavior is different between POSIX/XPG4 and ANSI. Refer to the *OS/390 C/C++ Run-Time Library Reference* for more information about requirements for each function.

You can invoke an OS/390 C/C++ program that uses OS/390 UNIX C functions using the following methods:

- Directly from the OS/390 UNIX Shell.
- From another program, or from the OS/390 UNIX Shell, using one of the `exec` family of functions, or the `BPXBATCH` utility from TSO or MVS batch.
- Using the `POSIX system()` call.
- Directly through TSO or MVS batch without the use of the intermediate `BPXBATCH` utility. In some cases, you may require the `POSIX(ON)` runtime option.

Input and Output

The C/C++ runtime library that supports the OS/390 C/C++ compiler supports different input and output (I/O) interfaces, file types, and access methods. The C++ I/O Stream Class Library provides additional support.

I/O Interfaces

The C/C++ runtime library supports the following I/O interfaces:

C Stream I/O

This is the default and the ANSI-defined I/O method. This method processes all input and output by character.

Record I/O

The library can also process your input and output by record. A record is a set of data that is treated as a unit. It can also process VSAM data sets by record. Record I/O is an OS/390 C/C++ extension to the ANSI standard.

TCP/IP Sockets I/O

OS/390 UNIX provides support for an enhanced version of an industry-accepted protocol for client/server communication that is known as *sockets*. A set of C language functions provides support for OS/390 UNIX sockets. OS/390 UNIX sockets correspond closely to the sockets that are used by UNIX applications that use the Berkeley Software Distribution (BSD) 4.3 standard (also known as OE sockets). The slightly different interface of the X/Open CAE Specification, Networking Services, Issue 4, is supplied as an additional choice. This interface is known as X/Open Sockets.

The OS/390 UNIX socket application program interface (API) provides support for both UNIX domain sockets and Internet domain sockets. UNIX domain sockets, or *local sockets*, allow interprocess communication within OS/390 independent of TCP/IP. Local sockets behave like traditional UNIX sockets and allow processes to communicate with one another on a single system. With Internet sockets, application programs can communicate with others in the network using TCP/IP.

In addition, the C++ I/O Stream Library supports formatted I/O in C++. You can code sophisticated I/O statements easily and clearly, and define input and output for your own data types. This helps improve the maintainability of programs that use input and output.

File Types

In addition to conventional files, such as sequential files and partitioned data sets, the C/C++ runtime library supports the following file types:

Virtual Storage Access Method (VSAM) Data Sets

OS/390 C/C++ has native support for three types of VSAM data organization:

- Key-sequenced data sets (KSDS). Use KSDS to access a record through a key within the record. A key is one or more consecutive characters that are taken from a data record that identifies the record.
- Entry-sequenced data sets (ESDS). Use ESDS to access data in the order it was created (or in the reverse order).
- Relative-record data sets (RRDS). Use RRDS for data in which each item has a particular number (for example, a telephone system with a record associated with each number).

For more information on how to perform I/O operations on these VSAM file types, see the *OS/390 C/C++ Programming Guide*.

Hierarchical File System Files

When you are running under MVS, TSO (batch and interactive), or IMS environments, OS/390 C/C++ recognizes a Hierarchical File System (HFS) file. The name specified on the `fopen()` or `freopen()` call has to conform to certain rules (described in the *OS/390 C/C++ Programming Guide*). You can create regular HFS files, special character HFS files, or FIFO HFS files. You can also create links or directories.

Memory Files

Memory files are temporary files that reside in memory. For improved performance, you can direct input and output to memory files rather than to devices. Since memory files reside in main storage and only exist while the program is executing, you primarily use them as work files. You can access memory files across load modules through calls to non-POSIX `system()` and `C fetch()`; they exist for the life of the root program. Standard streams can be redirected to memory files on a non-POSIX `system()` call using command line redirection.

Hiperspace* Expanded Storage

Large memory files can be placed in Hiperspace expanded storage to free up some of your home address space for other uses. Hiperspace expanded storage or high performance space is a range of up to 2 gigabytes of contiguous virtual storage space. A program can use this storage as a buffer (1 gigabyte = 2^{30} bytes).

Additional I/O Features

IBM OS/390 C/C++ provides additional I/O support through the following features:

- User error handling for serious I/O failures (SIGIOERR)
- Improved sequential data access performance through enablement of the DFSMS/MVS support for 31-bit sequential data buffers and sequential data striping on extended format data sets
- Full support of PDS/Es on OS/390 — including support for multiple members opened for write
- Overlapped I/O support under OS/390 (NCP, BUFNO)
- Multibyte character I/O functions
- Fixed-point (packed) decimal data type support in formatted I/O functions
- Support for multiple volume data sets that span more than one volume of DASD or tape
- Support for Generation Data Group I/O

The System Programming C Facility

The System Programming C (SP C) facility allows you to build applications that require no dynamic loading of OS/390 Language Environment libraries. It also allows you to tailor your application to better utilize the low-level services available on your operating system. SP C offers a number of advantages:

- You can develop applications that you can execute in a customized environment rather than with OS/390 Language Environment services. Note that if you do not use OS/390 Language Environment services, only some built-in functions and a limited set of C/C++ runtime library functions are available to you.
- You can substitute the OS/390 C language in place of assembler language when writing system exit routines, by using the interfaces that are provided by SP C.
- SP C lets you develop applications featuring a user-controlled environment, in which an OS/390 C environment is created once and used repeatedly for C function execution from other languages.
- You can utilize co-routines, by using a two-stack model to write application service routines. In this model, the application calls on the service routine to perform services independently of the user. The application is then suspended when control is returned to the user application.

Interaction with Other IBM Products

When you use OS/390 C/C++, you can write programs that utilize the power of other IBM products and subsystems:

- Cross System Product (CSP)

Cross System Product/Application Development (CSP/AD) is an application generator that provides ways to interactively define, test, and generate application programs to improve productivity in application development. Cross

System Product/Application Execution (CSP/AE) takes the generated program and executes it in a production environment.

Note: You cannot compile CSP applications with the OS/390 C++ compiler. However, your OS/390 C++ program can use interlanguage calls (ILC) to call OS/390 C programs that access CSP.

- Customer Information Control System (CICS)

You can use the CICS/ESA Command-Level Interface to write C/C++ application programs. The CICS Command-Level Interface provides data, job, and task management facilities that are normally provided by the operating system.

Note: Code preprocessed with CICS/ESA versions prior to V4 R1 is not supported for OS/390 C++ applications. OS/390 C++ code preprocessed on CICS/ESA V4 R1 cannot run under CICS/ESA V3 R3.

- DATABASE 2 (DB2)

DB2 programs manage data that is stored in relational data bases. The IBM DATABASE 2 licensed program runs on OS/390.

You can access the data by using a structured set of queries that are written in Structured Query Language (SQL). The DB2 program uses SQL statements that are embedded in the program. The SQL translator (DB2 preprocessor) translates the embedded SQL into host language statements that perform the requested functions. The OS/390 C/C++ compilers compile the output of the SQL translator. The DB2 program processes a request, and processing returns to the application.

- Data Window Services (DWS)

The Data Window Services (DWS) part of the Callable Services Library allows your OS/390 C or OS/390 C++ program to manipulate temporary data objects that are known as TEMPSPACE and VSAM linear data sets.

- Information Management System (IMS)

The Information Management System/Enterprise Systems Architecture (IMS/ESA) product provides support for hierarchical databases.

- Interactive System Productivity Facility (ISPF)

OS/390 C/C++ provides access to the Interactive System Productivity Facility (ISPF) Dialog Management Services. A dialog is the interaction between a person and a computer. The dialog interface contains display, variable, message, and dialog services as well as other facilities that are used to write interactive applications.

- Graphical Data Display Manager (GDDM)

GDDM provides a comprehensive set of functions to display and print applications most effectively:

- A windowing system that the user can tailor to display selected information
- Support for presentation and keyboard interaction
- Comprehensive graphics support
- Fonts — including support for double-byte character set (DBCS)
- Business image support

- Saving and restoring graphics pictures
- Support for many types of display terminals, printers, and plotters
- Query Management Facility (QMF)

OS/390 C supports the Query Management Facility (QMF), a query and report writing facility, which allows you to write applications through a callable interface. You can create applications to perform a variety of tasks, such as data entry, query building, administration aids, and report analysis.

Additional Features of OS/390 C/C++

Feature	Description
Multibyte Character Support	OS/390 C/C++ supports multibyte characters for those national languages such as Japanese whose characters cannot be represented by a single byte.
Wide Character Support	Multibyte characters can be normalized by OS/390 C library functions and encoded in units of one length. These normalized characters are called wide characters. Conversions between multibyte and wide characters can be performed by string conversion functions such as <code>wcstombs()</code> , <code>mbstowcs()</code> , <code>wcsrtoombs()</code> , and <code>mbsrtowcs()</code> , as well as the family of wide-character I/O functions. Wide-character data can be represented by the <code>wchar_t</code> data type.
Extended Precision Floating-Point Numbers	<p>OS/390 C/C++ provides three S/370 floating-point number data types: single precision (32 bits), declared as <code>float</code>; double precision (64 bits), declared as <code>double</code>; and extended precision (128 bits), declared as <code>long double</code>.</p> <p>Extended precision floating-point numbers give greater accuracy to mathematical calculations.</p> <p>As of Release 6, OS/390 C/C++ also supports IEEE 754 floating-point representation. By default, <code>float</code>, <code>double</code>, and <code>long double</code> values are represented in IBM S/390 floating point format. However, the IEEE 754 floating-point representation is used if you specify the <code>FLOAT(IEEE754)</code> compile option. For details on this support, see the description of the <code>FLOAT</code> option in the <i>OS/390 C/C++ User's Guide</i>.</p>
Command Line Redirection	You can redirect the standard streams <code>stdin</code> , <code>stderr</code> , and <code>stdout</code> from the command line or when calling programs using the <code>system()</code> function.
National Language Support	OS/390 C/C++ provides message text in either American English or Japanese. You can dynamically switch between the two languages.
Locale Definition Support	OS/390 C/C++ provides a locale definition utility that supports the creation of separate files of internationalization data, or locales. Locales can be used at run time to customize the behavior of an application to national language, culture, and coded character set (code page) requirements. Locale-sensitive library functions, such as <code>isdigit()</code> , use this information.
Coded Character Set (Code page) Support	The OS/390 C/C++ compiler can compile C/C++ source written in different EBCDIC code pages. In addition, the <code>iconv</code> utility converts data or source from one code page to another.
Selected Built-in Library Functions	Selected library functions, such as string and character functions, are built into the compiler to improve performance execution. Built-in functions are compiled into the executable, and no calls to the library are generated.
Multitasking Facility (MTF)	Multitasking is a mode of operation where your program performs two or more tasks at the same time. OS/390 C provides a set of library functions that perform multitasking. These functions are known as the Multitasking Facility (MTF). MTF uses the multitasking capabilities of OS/390 to allow a single OS/390 C application program to use more than one processor of a multiprocessing system simultaneously.

Feature	Description
Packed Structures and Unions	OS/390 C provides support for packed structures and unions. Structures and unions may be packed to reduce the storage requirements of a OS/390 C program.
Fixed-point (Packed) Decimal Data	OS/390 C supports fixed-point (packed) decimal as a native data type for use in business applications. The packed data type is similar to the COBOL data type COMP-3 or the PL/I data type FIXED DEC, with up to 31 digits of precision. The Application Support Class Library provides the Binary Coded Decimal Class for C++ programs.
Long Name Support	For portability, external names can be mixed case and up to 1024 characters in length. For C++, the limit applies to the mangled version of the name.
System Calls	You can call commands or executable modules using the <code>system()</code> function under OS/390, OS/390 UNIX, and TSO. You can also use the <code>system()</code> function to call EXECs on OS/390 and TSO, or Shell scripts using OS/390 UNIX.
Exploitation of ESA	Support for OS/390, IMS/ESA, Hiperspace expanded storage, and CICS/ESA allows you to exploit the features of the ESA.
Exploitation of hardware	Use the ARCHITECTURE compiler option to select the minimum level of machine architecture on which your program will run. ARCH(3) enables support for IEEE 754 Binary Floating-Point instructions. ARCH(2) instructs the compiler to generate faster instruction sequences available only on newer machines. Use the TUNE compiler option to optimize your application for selected machine architecture. TUNE(3) optimizes your application for the new G5 processor. TUNE(2) optimizes your application for other architectures. For information on which machines and architectures support the above options, refer to the ARCHITECTURE and TUNE compiler information in the <i>OS/390 C/C++ User's Guide</i> .

Suggested Reading

The following is a sample of some publications that are generally available. It is not an exhaustive list. Other publications may be available in your locality.

The Annotated C++ Reference Manual by Margaret A. Ellis and Bjarne Stroustrup, Addison-Wesley Publishing Company.

The C++ Programming Language (Second Edition) by Bjarne Stroustrup, Addison-Wesley Publishing Company.

C++ Primer (Second Edition) by Stanley B. Lippman, Addison-Wesley Publishing Company.

These books contain explanations of data structures that may help you understand the data structures in the Collection Classes:

Data Structures and Algorithms by Aho, Hopcroft, and Ullman, Addison-Wesley Publishing Company.

The Art of Computer Programming, Vol. 3: Sorting and Searching, D.E. Knuth, Addison-Wesley Publishing Company.

C++ Components and Algorithms by Scott Robert Ladd, M&T Publishing Inc.

A Systematic Catalogue of Reusable Abstract Data Types by Juergen Uhl and Hans Albrecht Schmit, Springer Verlag.

Part 1. Complex Mathematics Library

Chapter 1. complex Class	3
Constants Defined in complex.h	3
Constructors for complex	4
Mathematical Operators for complex	5
Input and Output Operators for complex	6
Mathematical Functions for complex	7
Trigonometric Functions for complex	8
Magnitude Functions for complex	8
Conversion Functions for complex	8
 Chapter 2. c_exception Class	11
Constructor for c_exception	11
Data Members of c_exception	11
Errors Handled by the Complex Mathematics Library	12

Chapter 1. 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	4	<code>conj</code>	9
operator <code>+</code>	5	<code>cos</code>	8
operator <code>+=</code> , <code>-=</code> , <code>*=</code> , <code>/=</code>	6	<code>cosh</code>	8
operator <code>!=</code>	6	<code>exp</code>	7
operator <code>*</code>	5	<code>imag</code>	9
operator <code>-</code> (negation)	5	<code>log</code>	7
operator <code>-</code> (subtraction)	5	<code>norm</code>	8
operator <code>/</code>	5	<code>polar</code>	9
operator <code>>></code>	6	<code>pow</code>	7
operator <code><<</code>	6	<code>real</code>	9
operator <code>==</code>	5	<code>sin</code>	8
<code>abs</code>	8	<code>sinh</code>	8
<code>arg</code>	9	<code>sqrt</code>	7

Constants Defined in `complex.h`

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

Table 3 (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>	$\pi / 2$
<code>M_PI_4</code>	$\pi / 4$
<code>M_1_PI</code>	$1 / \pi$
<code>M_2_PI</code>	$2 / \pi$
<code>M_2_SQRTPI</code>	2 divided by the square root of π

Table 3 (Page 2 of 2). Constants Defined in `complex.h`

Constant Name	Description
<code>M_SQRT2</code>	The square root of 2
<code>M_SQRT1_2</code>	The square root of 1 / 2

Constructors for complex

There are two versions of the complex constructor:

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

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

```
complex comp;
```

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

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

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

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

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

There is no explicit complex destructor.

Initializing complex Arrays

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

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

This example produces the following output:

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

Mathematical Operators for complex

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

Addition

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

The addition operator returns the sum of x and y .

Subtraction

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

The subtraction operator returns the difference between x and y .

Negation

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

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

Multiplication

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

The multiplication operator returns the product of x and y .

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 *IBM 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 *IBM Open Class Library User's Guide*.

Mathematical Assignment Operators

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

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

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

Note: The assignment operators do not produce a value that can be used in an expression. The following code, for example, produces a compile-time error:

```
complex x, y, z;      // valid declaration  
x = (y += z);         // invalid assignment causes a  
                      // compile-time error  
  
y += z;              // correct method involves splitting  
x = y;               // expression into separate statements
```

Input and Output Operators for complex

Input Operator

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

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

Output Operator

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

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

Mathematical Functions for complex

exp

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

`exp()` returns the complex value equal to e^x where x is the argument. Table 4 on page 13 shows the values returned by the default error-handling procedure for `exp()`.

log

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

`log()` returns the natural logarithm of the argument x . Table 4 on page 13 shows the values returned by the default error-handling procedure for `log()`.

pow

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

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

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

sqrt

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

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

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

has two square roots:

- (c,d)
- $(-c,-d)$

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

Trigonometric Functions for complex

cos

friend complex **cos**(complex *x*);

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

cosh

friend complex **cosh**(complex *x*);

`cosh()` returns the hyperbolic cosine of *x*. Table 4 on page 13 shows the values returned by the default error-handling procedure for `cosh()`.

sin

friend complex **sin**(complex *x*);

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

sinh

friend complex **sinh**(complex *x*);

`sinh()` returns the hyperbolic sine of *x*. Table 4 on page 13 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 5 in the *IBM 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) .

polar

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

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

real

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

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

imag

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

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

Chapter 2. 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	11	name	11
arg1	11	retval	12
arg2	11	type	12

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&`.

Note: You can only override `complex_error()` if you are using the static version of the I/O Stream Library.

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()`, you must ensure that the name of the header file that contains your version of `complex_error()` is included in your source file when you compile your program.

Default Error-Handling Procedures

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

Notes:

The following symbols appear in this table:

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

Table 4. 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)	EDOM (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
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: A message is also produced when `errno` is set to `EDOM`.

Part 2. I/O Stream Library

Chapter 3. filebuf Class	17
Public Members of filebuf	18
Chapter 4. fstream, ifstream, and ofstream Classes	23
Public Members of fstreambase	23
Public Members of fstream	24
Public Members of ifstream	26
Public Members of ofstream	28
Chapter 5. ios Class	31
Constructors and Assignment Operator for ios	32
Format State Variables	32
Format State Flags	33
Public Members of ios for the Format State	36
Public Members of ios for User-Defined Format Flags	38
Public Members of ios for the Error State	39
Other Members of ios	41
Built-In Manipulators for ios	42
Chapter 6. istream and istream_withassign Classes	43
Public Members of istream and istream_withassign	43
Chapter 7. ostream and ostream_withassign Classes	45
Constructors for ostream	45
Input Prefix Function	45
Public Members of ostream for Formatted Input	46
Public Members of ostream for Unformatted Input	49
Public Members of ostream for Positioning	51
Other Public Members of ostream	51
Built-In Manipulators for ostream	52
Public Members of ostream_withassign	53
Chapter 8. Manipulators	55
Parameterized Manipulators for the Format State	55
Chapter 9. ostream and ostream_withassign Classes	59
Constructors for ostream	59
Output Prefix and Suffix Functions	60
Public Members of ostream for Formatted Output	60
Public Members of ostream for Unformatted Output	63
Public Members of ostream for Positioning	64
Other Public Members of ostream	64
Built-In Manipulators for ostream	64
Public Members of ostream_withassign	65
Chapter 10. stdiobuf and stdiostream Classes	67
Public Members of stdiobuf	67
Public Members of stdiostream	68

Chapter 11. streambuf Class	71
streambuf Public and Protected Interfaces	71
Public Members of the streambuf Public Interface	72
Protected Functions That Return Pointers	75
Protected Functions That Set Pointers	76
Other Nonvirtual Protected Member Functions	77
Protected Virtual Member Functions	78
 Chapter 12. stringstream, istream, and ostream Classes	 83
Public Members of stringstream	83
Public Members of stringstream	84
Public Members of istream	84
Public Members of ostream	85
 Chapter 13. stringstream Class	 87
Public Members of stringstream	87

Chapter 3. 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	18	<code>is_open</code>	19
<code>filebuf</code> destructor	18	<code>open</code>	19
<code>attach</code>	18	<code>seekoff</code>	19
<code>close</code>	18	<code>seekpos</code>	20
<code>detach</code>	18	<code>setbuf</code>	20
<code>fd</code>	19	<code>sync</code>	20
<code>fp</code>	19		

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

Public Members of filebuf

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

Constructors for filebuf

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

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

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

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

Destructor for filebuf

```
~filebuf();
```

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

attach

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

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

On OS/390, if you have a file pointer already opened, use the following function to do the attach instead of using the file descriptor.

```
filebuf* attach(FILE *fp);
```

`attach()` attaches *fb* to the file pointer *fp*. If *fb* is already open, `attach()` returns 0. Otherwise, `attach()` returns a pointer to *fb*.

Note: This member is only supported under OS/390 C/C++.

detach

```
int detach();
```

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

close

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

`close()` does the following:

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

3. Closes the file that was attached to *fb*

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

fd

```
int fd();
```

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

fp

```
FILE* fp();
```

`fp()` returns the file pointer that is attached to *fb*. If *fb* is not opened, `fp()` returns 0.

Note: This member is only supported under OS/390 C/C++.

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);
filebuf* open(const char* fname, const char* fattr,
               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 second version of `open()` is specific to the OS/390 C/C++ implementation of C++. You can use the *fattr* parameter to specify additional file attributes, such as `lrecl` or `recfm`. All the parameters documented for the `fopen()` function in the *OS/390 C/C++ Run-Time Library Reference* and *OS/390 C/C++ Programming Guide* are supported, with the exception of `type=record`.

Note: The *prot* parameter is ignored on OS/390 C/C++.

seekoff

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

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

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

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

If `fb` is attached to a file that does not support seeking, or if the value `sd + so` specifies a position before the beginning of the file, `seekoff()` returns `E0F` and the position of the file get/put pointer is undefined. Otherwise, `seekoff()` returns the new position of the file get/put pointer.

You can use relative byte offsets when seeking from `ios::cur` or `ios::end`. You can use relative byte offsets when seeking from `ios::beg` if either of the following conditions are true:

- The file is not a variable record format file, and is opened for binary I/O
- The file is a variable record format file, and is opened for binary I/O with the `bytesseek` option. The `bytesseek` option is enabled for a specific file if the `bytesseek` `fopen()` option is passed when the file is opened (see the MVS-specific constructors and `open()` functions in this book, and `fopen()` in the *OS/390 C/C++ Run-Time Library Reference*). The `bytesseek` option can also be enabled for all files if you set the `_EDC_BYTESEEK` environment variable (see the *OS/390 C/C++ Programming Guide*).

When seeking from `ios::beg` in text files, encoded offsets are used. You can only seek to an offset value returned by a previous call to `seekoff()`, and attempting to calculate a new position based on an encoded offset value results in undefined behaviour.

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.

Chapter 4. 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	24	constructor	26
close	24	open	27
detach	24	rdbuf	27
setbuf	24	ofstream:	
fstream:		constructor	28
constructor	24	open	28
open	25	rdbuf	29
rdbuf	26		

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*.

```
void attach(FILE *fp);
```

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

Note: This member is only supported under OS/390 C/C++.

close

```
void close();
```

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

detach

```
int detach();
```

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

setbuf

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

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

Public Members of 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);
```

This 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 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.

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

This version is specific to the OS/390 C/C++ implementation of C++. You can use the *fattr* parameter to specify additional file attributes, such as `lrecl` or `recfm`. All the parameters documented for the `fopen()` function in the *OS/390 C/C++ Run-Time Library Reference* and *OS/390 C/C++ Programming Guide* are supported, with the exception of `type=record`.

Note: The *prot* attribute is ignored on OS/390 C/C++.

open

```
void open(const char* fname, int mode, int prot=filebuf::openprot);
void open(const char* fname, const char* fattr,
        int omode, 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 second version of `open()` is specific to the OS/390 C/C++ implementation of C++. You can use the *fattr* parameter to specify additional file attributes, such as `lrecl` or `recfm`. All the parameters documented for the `fopen()` function in the *OS/390 C/C++ Run-Time Library Reference* and *OS/390 C/C++ Programming Guide* are supported, with the exception of `type=record`.

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.

ios::binary	The file is opened in binary mode. In the default (text) mode, carriage returns are discarded on input. (0x1a) 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 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 *IBM Open Class Library User's Guide*.

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

Constructors for ifstream

```
ifstream();
```

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

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

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

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

The third version constructs an `ifstream` object and opens the file *fname* with open mode equal to *mode* and protection mode equal to *prot*. The default value for

mode is `ios::in`, and the default value for *prot* is `filebuf::openprot`. If the file cannot be opened, the error state of the constructed `ifstream` object is set.

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

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

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

This version is specific to the OS/390 C/C++ implementation of C++. You can use the *fattr* parameter to specify additional file attributes, such as `lrecl` or `recfm`. All the parameters documented for the `fopen()` function in the *OS/390 C/C++ Run-Time Library Reference* and *OS/390 C/C++ Programming Guide* are supported, with the exception of `type=record`.

Note: The *prot* attribute is ignored on OS/390 C/C++.

open

```
void open(const char* fname,
          int mode=ios::in,
          int prot=filebuf::openprot);
void open(const char* fname, const char* fattr,
          int omode=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 second version of `open()` is specific to the OS/390 C/C++ implementation of C++. You can use the *fattr* parameter to specify additional file attributes, such as `lrecl` or `recfm`. All the parameters documented for the `fopen()` function in the *OS/390 C/C++ Run-Time Library Reference* and *OS/390 C/C++ Programming Guide* are supported, with the exception of `type=record`.

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 25 for a list of the possible values for *mode*.

Note: The *prot* attribute is ignored on OS/390 C/C++.

rdbuf

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

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

Public Members of ofstream

For an example of using the ofstream class, see “Opening a File for Output and Writing to the File” in the *IBM 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.

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

This version is specific to the OS/390 C/C++ implementation of C++. You can use the *fattr* parameter to specify additional file attributes, such as lrecl or recfm. All the parameters documented for the fopen() function in the *OS/390 C/C++ Run-Time Library Reference* and *OS/390 C/C++ Programming Guide* are supported, with the exception of type=record.

Note: The prot attribute is ignored on OS/390 C/C++.

open

```
void open(const char* fname, int mode, int prot=filebuf::openprot);
void open(const char* fname, const char* fattr,
          int omode, 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 second version of `open()` is specific to the OS/390 C/C++ implementation of C++. You can use the *fattr* parameter to specify additional file attributes, such as `lrecl` or `recfm`. All the parameters documented for the `fopen()` function in the *OS/390 C/C++ Run-Time Library Reference* and *OS/390 C/C++ Programming Guide* are supported, with the exception of `type=record`.

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

The members of the `ios::open_mode` enumeration are bits that can be 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 25

Note: The *prot* attribute is ignored on OS/390 C/C++. 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*.

Chapter 5. 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	32	<code>precision</code>	36
<code>bad</code>	39	<code>pword</code>	38
<code>bitalloc</code>	38	<code>rdbuf</code>	41
<code>clear</code>	39	<code>rdstate</code>	40
<i><code>dec</code></i>	34	<i><code>right</code></i>	34
<code>dec</code> manipulator	42	<i><code>scientific</code></i>	34
<code>endl</code> manipulator	42	<code>setf</code>	37
<code>ends</code> manipulator	42	<i><code>showbase</code></i>	34
<code>eof</code>	39	<i><code>showpoint</code></i>	34
<code>fail</code>	40	<i><code>showpos</code></i>	34
<code>fill</code>	36	<code>skip</code>	37
<i><code>fixed</code></i>	35	<i><code>skipws</code></i>	33
<code>flags</code>	36	<i><code>stdio</code></i>	35
<code>flush</code> manipulator	42	<code>sync_with_stdio</code>	41
<code>good</code>	40	<code>tie</code>	41
<i><code>hex</code></i>	34	<i><code>unitbuf</code></i>	35
<code>hex</code> manipulator	42	<code>unsetf</code>	37
<i><code>internal</code></i>	34	<i><code>uppercase</code></i>	35
<code>iword</code>	38	<code>width</code>	37
<i><code>left</code></i>	33	<code>ws</code> manipulator	42
<i><code>oct</code></i>	34	<i><code>x_fill</code></i>	32
<code>oct</code> manipulator	42	<i><code>x_precision</code></i>	32
<code>operator void*</code>	40	<i><code>x_width</code></i>	33
<code>operator=</code>	32	<code>xalloc</code>	39

Constructors and Assignment Operator for ios

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

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

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

Copying of `ios` objects is not well defined, and for this reason, both the assignment operator and the copy constructor are declared `private`. Assignment between streams is supported by the `istream_withassign`, `ostream_withassign`, and `iostream_withassign` classes. See “Assignment Operator for `istream_withassign`” on page 53 and “Assignment Operator for `ostream_withassign`” on page 65 for more details. Except for the `..._withassign` classes, none of the predefined classes derived from `ios` has a copy constructor or an assignment operator. Unless you define your own copy constructor or assignment operator for a class that you derive from `ios`, your class will have neither a copy constructor nor an assignment operator.

Format State Variables

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

x_fill

```
char x_fill;
```

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

x_precision

```
short x_precision;
```

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

x_width

```
short x_width;
```

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

Format State Flags

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

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

“Mutually Exclusive Format Flags” on page 35 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.

```
#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.

Format State Flags

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 42 for more details) have the same effect as the flags `ios::dec`, `ios::oct`, and `ios::hex`, respectively. `dec` is set by default.

dec

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

oct

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

hex

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

showbase

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

Integral Formatting

showpos

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

Floating-Point Formatting

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

showpoint

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

scientific

If `ios::scientific` is set, the value is converted using scientific notation. In scientific notation, there is one digit before the decimal point and the number of digits following the decimal point depends on the value of `ios::x_precision`. The default value for `ios::x_precision` is 6. If `ios::uppercase` is set, an uppercase “E” precedes the exponent. Otherwise, a lowercase “e” precedes the exponent. By default, uppercase is not set. See “uppercase” on page 35 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` are inserted in uppercase.

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 41 for more details on `sync_with_stdio()`. `ios::stdio` is not set by default.

Mutually Exclusive Format Flags

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

- `ios::left`, `ios::right`, `ios::internal`
- `ios::dec`, `ios::oct`, `ios::hex`

- `ios::scientific`, `ios::fixed`

Public Members of ios for the Format State

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

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

fill

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

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

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

You can also use the parameterized manipulator `setfill` to set the value of `ios::x_fill`. See “setfill” on page 56 for a description of this parameterized manipulator.

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 57 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 56 for more details on this parameterized manipulator.

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

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

In this statement, `ios::basefield` specifies the conversion base as the format flag that is going to be changed, and `ios::hex` specifies the new value for the conversion base. If *newset* equals 0, all of the format flags specified in *field* are cleared. You can also use the parameterized manipulator `resetiosflags` to clear format flags. See “`resetiosflags`” on page 56 for more details on this parameterized manipulator.

Note: If you set conflicting flags the results are unpredictable. See “Mutually Exclusive Format Flags” on page 35 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

User-Defined Format Flags

`ios::x_width` is 0, the operations that insert values only insert the characters needed to represent a value.

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

You can also use the parameterized manipulator `setw` to set the field width. See “`setw`” on page 57 for more information on this parameterized manipulator. Also, see “Public Members of `ostream` for Formatted Output” on page 60 for more information on `ios::x_width`.

Public Members of `ios` for User-Defined Format Flags

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

bitalloc

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

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

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**.

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 32 for more details on the `ios` constructor.

sync_with_stdio

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

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

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

`sync_with_stdio()` makes `cout` and `clog` unit buffered. See “Buffer Flushing” on page 35 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.

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.

Built-In Manipulators

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>

Chapter 6. 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	43
<code>iostream_withassign</code> Constructor	43
<code>iostream_withassign</code> Assignment Operator	43

Public Members of `iostream` and `iostream_withassign`

Constructor for `iostream`

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

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

Constructor for `iostream_withassign`

```
iostream_withassign();
```

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

Assignment Operator for `iostream_withassign`

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

There are two versions of the `iostream_withassign` assignment operator. The first version takes a reference to an `ios` object, `is`, as its argument. It associates the

stream buffer attached to *is* with the `iostream_withassign` object that is on the left side of the assignment operator.

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

Chapter 7. 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>	45	<code>tellg</code>	51
<code>istream</code> Constructor	45	<code>gcount</code>	51
<code>input</code> operator	46	<code>peek</code>	51
<code>get</code>	49	<code>putback</code>	52
<code>getline</code>	50	<code>sync</code>	52
<code>ignore</code>	50	<code>istream_withassign</code> Constructor	53
<code>read</code>	50	<code>istream_withassign</code> operator=	53
<code>seekg</code>	51		

Constructors for istream

Constructor for istream

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

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

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

Input Prefix Function

```
int ipfx(int need=0);
```

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

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

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

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

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

Public Members of `istream` for Formatted Input

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

```
ins >> x;
```

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

The details of conversion depend on the format state (see “Format State Variables” on page 32 for details) of the `istream` object and the type of the variable *x*. The input operator may set the width variable `ios::x_width` to 0, but it does not change anything else in the format state. See “Input Operator for Arrays of Characters” on page 47 below for details.

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`, `long double`, and `long long` 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 *IBM 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.

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);
```

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.
- `ios::hex`: the characters are converted to a hexadecimal value. Characters are extracted from the stream buffer until a character that is not a hexadecimal digit (a digit from 0 to 9 or a letter from “A” to “F”, upper or lower case) is encountered. If `ios::hex` is set and a signed value is encountered, the value is converted into a decimal value. For example, if the characters “-12” are encountered in the input stream and `ios::hex` is set, the decimal value -18 is actually extracted.

If none of these format flags is set, the characters are converted according to the C++ lexical conventions.

This conversion depends on the characters that follow the optional sign:

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

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

Input Operator for float and double Values

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

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

Note that the `istream` input operator on some other operating systems converts the strings `nan` and `infinity` (in any combination of upper- and lowercase letters) into numeric representations of not-a-number and infinity. If you use these string values as input in a program compiled with OS/390 C/C++, the input operator will not

recognize them as floating point numbers and will set `ios::badbit` in the stream's error state.

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

Input Operator for `streambuf` Objects

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

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

Public Members of `istream` for Unformatted Input

You can use the functions listed in this section to extract characters from a stream buffer as a sequence of bytes. All of these functions call `ipfx(1)`. They only proceed with their processing if `ipfx(1)` returns a nonzero value. See “Input Prefix Function” on page 45 for more details on `ipfx()`.

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

`get`

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

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

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

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

`get`

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

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

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

get

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

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

get

```
int get();
```

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

getline

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

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

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

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

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

See “White Space in String Input” in the *IBM 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.

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 49 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 46 for more details on formatted input.

peek

```
int peek();
```

`peek()` calls `ipfx(1)`. If `ipfx()` returns zero, or if no more input is available from the ultimate producer, `peek()` returns `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 45 for more details on `ipfx()`.

sync

```
int sync();
```

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

Built-In Manipulators for istream

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

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

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

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

```
ins >> ws;
```

This statement sets `ios::dec`:

```
ins >> dec;
```

This statement sets `ios::hex`:

```
ins >> hex;
```

This statement sets `ios::oct`:

```
ins >> oct;
```

Public Members of istream_withassign

Constructor for istream_withassign

```
istream_withassign();
```

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

Assignment Operator for istream_withassign

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

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

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

Chapter 8. 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 *IBM Open Class Library User's Guide* for further information.

Header File

The parameterized manipulator classes are declared in `iosmanip.h`.

Members

The following parameterized manipulators are described:

Manipulator	Page	Manipulator	Page
resetiosflags	56	setiosflags	56
setbase	56	setprecision	57
setfill	56	setw	57

Parameterized Manipulators for the Format State

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

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

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

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

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

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

resetiosflags

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

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

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

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

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

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

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

setbase

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

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

setfill

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

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

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

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

setiosflags

SMANIP(long) **setiosflags**(long *flags*);

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

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

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

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

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

setprecision

SMANIP(int) **setprecision**(int *prec*);

`setprecision()` sets the precision format state variable, `ios::x_prec`, to the value of *prec*. The value of *prec* must be greater than zero. If the value of *prec* is negative, the precision format state variable is set to 6. See “precision” on page 36 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 37 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);
```

Chapter 9. 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	59	osfx	60
output operator	61	put	63
ostream_withassign constructor	65	seekp	64
ostream_withassign operator=	65	tellp	64
flush	64	write	63
opfx	60		

Constructors for ostream

Constructor for ostream

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

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

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

Output Prefix and Suffix Functions

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

opfx

```
int opfx();
```

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

osfx

```
void osfx();
```

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

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

Public Members of `ostream` for Formatted Output

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

```
outs << x;
```

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

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

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, long double, and long long 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 *IBM Open Class Library User's Guide* for instructions on how to do this.

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

Output Operator for Arrays of Characters and char Values

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

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

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

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

Output Operator for Other Integral Values

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

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:

- `outs.precision()`: returns the number of significant digits that appear after the decimal. The default value is 6.
- `outs.width()`: if this returns 0, *val* is inserted without any fill characters. (See "fill" on page 36 for more details on fill characters.) If the return value is greater than the number of characters needed to represent *val*, extra fill characters are inserted so that the total number of characters inserted is equal to the return value.

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

- If `ios::scientific` is set, *val* is converted to scientific notation, with one digit before the decimal, and the number of digits after the decimal equal to the value returned by `outs.precision()`. The exponent begins with a lowercase

“e” unless `ios::uppercase` is set, in which case the exponent begins with an uppercase “E.”

- If `ios::fixed` is set, *val* is converted to fixed notation, with the number of digits after the decimal point equal to the value returned by `outs.precision()`. If neither `ios::fixed` nor `ios::scientific` is set, the conversion depends upon the value of *val*. See “Floating-Point Formatting” on page 34 for more details.
- If `ios::uppercase` is set, the exponents of values in scientific notation begin with an uppercase “E.”

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

Output Operator for Pointers to void

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

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

Output Operator for streambuf Objects

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

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

Public Members of ostream for Unformatted Output

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

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

put

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

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

write

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

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

Public Members of ostream for Positioning

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

seekp

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

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

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

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

tellp

```
streampos tellp();
```

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

Other Public Members of ostream

flush

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

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

Built-In Manipulators for ostream

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

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

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

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

```
outs << endl;
```

This statement inserts a null character:

```
outs << ends;
```

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

```
outs << flush;
```

This statement sets `ios::dec`:

```
outs << dec;
```

This statement sets `ios::hex`:

```
outs << hex;
```

This statement sets `ios::oct`:

```
outs << oct;
```

Public Members of ostream_withassign

Constructor for ostream_withassign

```
ostream_withassign();
```

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

Assignment Operator for ostream_withassign

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

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

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

Chapter 10. 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 Chapter 4, “`fstream`, `ifstream`, and `ofstream` Classes” on page 23 and Chapter 3, “`filebuf` Class” on page 17 for more details on these classes. See “`sync_with_stdio`” on page 41 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 `stdiostream.h`.

Members

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

Member	Page	Member	Page
stdiobuf		stdiostream	
Constructor	67	Constructor	68
Destructor	68	<code>rdbuf</code>	68
<code>stdiofile</code>	68		

Public Members of `stdiobuf`

Constructor for `stdiobuf`

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

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

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

Destructor for stdiobuf

```
~stdiobuf();
```

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

stdiofile

```
FILE* stdiofile();
```

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

Public Members of stdiostream
Constructor for stdiostream

```
stdiostream(FILE* file);
```

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

rdbuf

```
stdiobuf* rdbuf();
```

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

Example of Using stdiostream

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

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

void main()
{
    FILE *in = fopen("in.dat", "r");
    FILE *out = fopen("out.dat", "w");
    int c;
    if (in == NULL)
    {
        cerr << "Cannot open file 'in.dat' for reading."
              << endl;
        exit(1);
    }
    if (out == NULL)
    {
        cerr << "Cannot open file 'out.dat' for writing."
              << endl;
        exit(1);
    }
    //
    // Create a stdiostream object attached to "f"
    //
    stdiostream sin(in);
    stdiostream sout(out);
    cout << "The data contained in the file is: " << endl;
    //
```

```

// Now read data from "sin" and copy it to
// "cout" and "sout"
//
while ((c = sin.rdbuf()->sbumpc()) != EOF)
{
    cout << char(c);
    sout.rdbuf()->sputc(c);
}
cout << endl;
}

```

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

```
input input input input
```

The following output is produced:

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

Chapter 11. streambuf Class

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

Derivation

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

Header File

streambuf is declared in iostream.h.

Members

The following members are provided for streambuf:

Method	Page	Method	Page
streambuf constructors	73	pptr	76
streambuf destructor	73	sbumpc	73
allocate	77	seekoff	80
base	75	seekpos	80
blen	77	setb	76
dbp	77	setbuf	81
doallocate	79	setg	76
eback	75	setp	76
ebuf	75	sgetc	73
egptr	75	sgetn	74
epptr	75	snextc	74
gbump	78	sputbackc	74
gptr	75	sputc	74
in_avail	73	sputn	74
out_waiting	73	stossc	74
overflow	79	sync	81
pbackfail	79	unbuffered	78
pbase	75	underflow	81
pbump	78		

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 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()`, `eptr()`, `pptr()`, `gptr()`, and `pbase()` are initially all zero for this stream buffer.

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

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

Destructor for `streambuf`

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

The destructor for `streambuf` calls `sync()`. If a stream buffer has been set up and `ios::alloc` is set, `sync()` deletes the stream buffer. See “`sync`” on page 81 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.

`out_waiting`

```
int out_waiting();
```

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

`sbumpc`

```
int sbumpc();
```

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

`sgetc`

```
int sgetc();
```

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

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

sgetn

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

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

snextc

```
int snextc();
```

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

sputbackc

```
int sputbackc(char c);
```

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

sputc

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

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

sputn

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

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

stossc

```
void stossc();
```

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

Protected Functions That Return Pointers

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

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

base

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

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

eback

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

`eback()` returns a pointer to the lower bound of the space available for the get area of *dsb*. The space between the pointer returned by `eback()` and the pointer returned by `gptr()` is available for *putback*. See “putback” on page 52 for details on *putback*.

ebuf

```
char* ebuf();
```

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

egptr

```
char* egptr();
```

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

epptr

```
char* epptr();
```

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

gptr

```
char* gptr();
```

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

pbase

```
char* pbase();
```

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

pptr

```
char* pptr();
```

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

Protected Functions That Set Pointers

This section describes the functions in the protected interface of `streambuf` that set the boundaries of areas in a stream buffer. The values of these boundaries are returned by the functions described in “Protected Functions That Return Pointers” on page 75.

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 7 on page 32 in the *IBM Open Class Library User's Guide* for details on where each of these functions points to within the stream buffer.

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

Other Nonvirtual Protected Member Functions

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

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

allocate

```
int allocate();
```

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

- 0, if *dsb* already has a stream buffer set up (that is, *dsb*→`base()` returns a nonzero value), or if `unbuffered()` returns a nonzero value. (See “unbuffered” on page 78 for more details.) `allocate()` does not do any further processing if it returns 0.
- 1, if `allocate()` does set up a stream buffer.
- EOF, if the attempt to allocate space for the stream buffer fails.

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

blen

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

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

dbp

```
void dbp();
```

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

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

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

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

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

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

Virtual Member Functions

Here is some sample output.
buf at 0x20048100, base=0x20049000, ebuf=0x20049400,
pptr=0x20049000, ep_ptr=0x20049400, eback=0x0, gp_ptr=0x0, eg_ptr=0x0

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 75 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 76 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 77 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 77 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 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 78 for more details on `unbuffered()`. See “base” on page 75 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.

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 75 for more details on `eback()`. See “gptr” on page 75 for more details on `gptr()`.

The default definition of `pbackfail()` returns `E0F`.

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 `E0F`.

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.

seekpos

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

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

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

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

setbuf

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

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

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

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

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

sync

```
virtual int sync();
```

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

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

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

Otherwise, `sync()` returns EOF.

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

underflow

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

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

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

If you define `underflow()` in a class derived from `streambuf`, it should return the first character in the get area if the get area is not empty. If the get area is empty, `underflow()` should create a get area that is not empty and return the next character. If no more characters are available in the ultimate producer, `underflow()` should return `E0F` and leave the get area empty.

Chapter 12. strstream, istrstream, and ostrstream Classes

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

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

Derivation

```

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

Header File

`strstream`, `istrstream`, and `ostrstream` are declared in `strstream.h`.

Members

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

Method	Page	Method	Page
<code>istrstream</code> constructors	84	<code>strstream</code> destructor	84
<code>istrstream</code> destructor	85	<code>pcount</code>	86
<code>ostrstream</code> constructors	85	<code>rdbuf</code>	83
<code>ostrstream</code> destructor	85	<code>str</code> (<code>strstream</code>)	84
<code>strstream</code> constructor	84	<code>str</code> (<code>ostrstream</code>)	86

Public Members of strstreambase

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

rdbuf

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

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

Public Members of strstream

Constructor for strstream

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

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

The version of the strstream constructor that takes three arguments specifies that characters should be extracted and inserted into the array of bytes that starts at the position pointed to by *cp* with a length of *len* bytes. If *ios::ate* or *ios::app* is set in *mode*, *cp* points to a null-terminated string and insertions begin at the null character. Otherwise, insertions begin at the position pointed to by *cp*. You can use the *istream::seekg()* function to reposition the get pointer anywhere in this array. See “seekg” on page 51 for more details on seekg().

Destructor for strstream

```
~strstream();
```

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

str

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

str() returns a pointer to the stream buffer attached to the strstream and calls freeze() (see “freeze” on page 88) with a nonzero value to prevent the stream buffer from being deleted. If the stream buffer was constructed with an explicit array, the value returned is a pointer to that array. If the stream buffer was constructed in dynamic mode, *cp* points to the dynamically allocated area.

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

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

Public Members of istrstream

Constructors for istrstream

```
istrstream(char* cp);
istrstream(signed char* cp);
istrstream(unsigned char* cp);
istrstream(const char* cp);
istrstream(const signed char* cp);
istrstream(const unsigned char* cp);
```

```

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 51 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.

Public Members of `ostrstream`

Constructors for `ostrstream`

```

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

```

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

The versions of the `ostrstream` constructor that take three arguments specify that the stream buffer that is attached to the `ostrstream` 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 51 for more details on `seekg()`.

Destructor for `ostrstream`

```
~ostrstream();
```

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

ostream

str

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

`str()` returns a pointer to the stream buffer attached to the `ostream` and calls `freeze()` (see “freeze” on page 88) 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.

Chapter 13. strstreambuf Class

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

Derivation

streambuf
 strstreambuf

Header File

strstreambuf is declared in strstream.h.

Members

The following members are provided for strstreambuf:

Method	Page	Method	Page
strstreambuf constructors	87	seekoff	89
strstreambuf destructors	88	setbuf	90
doallocate	88	str	89
freeze	88	underflow	90
overflow	89		

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 90 for more details on setbuf().

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

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

used to allocate space. *free* is a pointer to the function that is used to free space. *free* is passed an argument that is a pointer to the array of bytes that *alloc* allocated. If *free* has a value of 0, the operator `delete` is used to free space.

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

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

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

Destructor for `strstreambuf`

```
~strstreambuf();
```

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

- If you created the `strstreambuf` object using the constructor that takes two pointers to functions as arguments (see “Constructors for `strstreambuf`” on page 87 for more details), the destructor frees the space allocated by the destructor by calling the function pointed to by the second argument to the constructor.
- If you created the `strstreambuf` object using any of the other constructors, the destructor calls the `delete` operator to free the space allocated by the constructor.

`doallocate`

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

`doallocate()` attempts to allocate space for a stream buffer. If you created the `strstreambuf` object using the constructor that takes two pointers to functions as arguments (see “Constructors for `strstreambuf`” on page 87 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.

If you constructed the `strstreambuf` in dynamic mode (see “Constructors for `strstreambuf`” on page 87), 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.

Chapter 14, "Introduction to Flat Collections" on page 93 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 94.

Chapter 14. 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 Set, is really `ISet(...)`, because the default implementation variant of a set is `ISet`.

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 Set as B* Tree, substitute `ISetAsBstTree<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 *IBM Open Class Library User's Guide* describes the purpose of the equality operation and ordering relation.) Where both equality operation and ordering relation are provided, the Collection Class Library may use either to determine element equality.

given ...

Refers to an argument of the described function, such as given element, given key, or given collection.

iteration order

The order in which elements are visited in `allElementsDo()` and `setToNext()` or `setToPrevious()`.

In ordered collections, the element at position 1 will be visited first, then the element at position 2, and so on. Sorted collections, in particular, are visited following the ordering relation provided for the element type.

In collections that are not ordered, the elements are visited in an arbitrary order. Each element is visited exactly once.

positioning property

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

same key	Refers to equality of keys as defined by the equality operation or ordering relation provided for the key type. Where both equality operation and ordering relation are provided, the Collection Class Library may use either to determine key equality.
this collection	The collection to which a function is applied. Contrast with the <i>given</i> collection, which is an argument supplied to a function. <i>The collection</i> is synonymous with <i>this collection</i> .
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:
 - 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.
- 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 *IBM 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.

Chapter 15. 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 99.

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.

operator!=

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

Returns true if the given collection is not equal to the collection. For a definition of equality for collections, see “operator==” on page 98.

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 99.

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 notification send a `modifyId` notification.

Return Value: Returns a reference to the collection.

Exceptions

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

operator==

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

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

Type of Collection	Condition
Unique Elements	If the collections have unique elements, any element that occurs in one collection must occur in the other collection.
Non-Unique Elements	If an element has <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.

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 106 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 notifications send an `addId` 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

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 notifications send a `modifyId` notification.

Exceptions

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

addAsFirst

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

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

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

Preconditions

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

Side Effects

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

Exceptions

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

addAsNext

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

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

Preconditions

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

Side Effects

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

Exceptions

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

addAtPosition

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

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

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

Preconditions

- The cursor must belong to the collection.
- $(1 \leq position \leq numberOfElements + 1)$.
- 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 notifications send an `addId` 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 108.

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 notifications send a `modifyId` 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 110.

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 notifications send a `modifyId` notification.

Exceptions

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

addOrReplaceElementWithKey

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

```
IBoolen 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 notifications send a `replaceId` 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 124.

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 notifications send a `modifyId` notification.

Exceptions

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

allElementsDo

```

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

IBoolean allElementsDo (
    IBoolean (*function) (Element const&, void*),
    void* additionalArgument = 0 ) const;

```

Calls the given function for all elements in the collection until the given function returns false. The elements are visited in iteration order. Additional arguments can be passed to the given function using *additionalArgument*. The additional argument defaults to zero if no additional argument is given.

Notes:

1. The given function must not remove elements from or add them to the collection. If you want to remove elements, you can use the `removeAll()` function with a property argument. For further information see “removeAll” on page 118.
2. For the non-**const** version of `allElementsDo()`, the given function must not manipulate the element in the collection in a way that changes the positioning property of the element.

Return Value: Returns true if the given function returns true for every element it is applied to. Returns false if the collection is empty or if the given function returns false for at least one 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 *IBM 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 118.
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. Returns false if the collection is empty or if the `applyTo()` function returns false for at least one element it is applied to.

any

```
Element const& any ( ) const;
```

Returns a reference to an arbitrary element of the collection.

Precondition: The collection must not be empty.

Exception: `IEmptyException`

compare

```
long compare ( CLASS_NAME const& collection,  
              long (*comparisonFunction)  
                (Element const& element1, Element const& element2)  
              ) const;
```

Compares the collection with the given collection. Comparison yields <0 if the collection is less than the given collection, 0 if the collection is equal to the given collection, and >0 if the collection is greater than the given collection. Comparison is defined by the first pair of corresponding elements, in both collections, that are not equal. If such a pair exists, the collection with the greater element is the greater one. Otherwise, the collection with more elements is the greater one.

Notes:

1. The given comparison function must return a result according to the following rules:

>0	if (element1 > element2)
0	if (element1 == element2)
<0	if (element1 < element2)
2. For elements of type `char*`, `compare()` is not locale-sensitive by default. Because it uses `strcmp()` and not `strcoll()`, it compares the binary values representing the characters, and is not based on the `LC_COLLATE` category of the current locale. Its results are reliable only for code pages and character sets in which the collating sequence matches the sequence of binary representations. 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 *IBM Open Class Library User's Guide*.

Return Value: Returns the result of the collection comparison.

contains

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

Returns true if the collection contains an element equal to the given element.

containsAllFrom

```
IBoollean containsAllFrom (  
    CLASS_NAME const& collection ) const;  
  
IBoollean containsAllFrom (  
    IACollection <Element> const& collection ) const;
```

Returns true if all the elements of the given collection are contained in the collection. The definition of containment is described in "contains."

containsAllKeysFrom

```

IBoolean containsAllKeysFrom (
    CLASS_NAME const& collection ) const;

IBoolean containsAllKeysFrom (
    IACollection <Element> const& collection ) const;

```

Returns true if all of the keys of the given collection are contained in the collection.

containsElementWithKey

```

IBoolean containsElementWithKey ( Key const& key ) const;

```

Returns true if the collection contains an element with the same key as the given key.

copy

```

void copy ( IACollection <Element> const& collection ) ;

```

Copies the given collection to this collection. `copy()` removes all elements from this collection, and adds the elements from the given collection. For information on how adding is done, see “addAllFrom” on page 99.

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 notifications send a `modifyId` 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.

deque

```
void deque ( ) ;  
  
void deque ( 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 notifications send a `modifyId` notification.

Exception: `IEmptyException`

differenceWith

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

Makes the collection the difference between the collection and the given collection. The *difference* of A and B (A minus B) is the set of elements that are contained in A but not in B.

The following rule applies for bags with duplicate elements: If bag P contains the element X *m* times and bag Q contains the element X *n* times, the *difference* of P and Q contains the element X *m-n* times if *m* > *n*, and zero times if *m* ≤ *n*.

Side Effects

- If any elements were removed, all cursors of this collection become undefined.
- If the element is removed, collection classes supporting notifications send a `modifyId` notification.

disableNotification

```
INotifier&  
    disableNotification();
```

Stops the notifier from sending notifications to its observers.

elementAt

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

Returns a reference to the element pointed to by the given cursor.

Note: For the version of `elementAt()` *without* the **const** suffix, do not manipulate the element or the key of the element in the collection in a way that changes the positioning property of the element.

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

Exception: `ICursorInvalidException`

elementAtPosition

```
Element const& elementAtPosition ( IPosition position ) const;
```

Returns a reference to the element at the given position in the collection.

Position 1 specifies the first element.

Position must be a valid position in the collection; that is,
 $(1 \leq \textit{position} \leq \text{numberOfElements}())$.

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`

enableNotification

```
INotifier&  
  enableNotification( IBoolean = true );
```

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.

enqueue

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

Adds the element to the collection, and sets the cursor to the added element. For ordinary queues, the given element is added as the last element. For priority queues, the element is added at a position determined by the ordering relation provided for the element or key type.

Preconditions

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

Side Effects

- All cursors of this collection except the given cursor become undefined.
- If the element is added, collection classes supporting notifications send a `modifyId` notification.

Exceptions

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

first

Element const& **first** () 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

isBounded

IBoolean **isBounded** () const;

Returns true if the collection is bounded.

isEmpty

IBoolean **isEmpty** () const;

Returns true if the collection is empty.

isEnabledForNotification

```

IBoolean
  isEnabledForNotification() const;

```

Returns true if a notifier can send notifications to its observers.

isFirstAt

```

IBoolean isFirstAt ( 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())`.

isLastAt

```

IBoolean isLastAt ( 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.

last

```

Element const& last ( ) const;

```

Returns a reference to the last element of the collection.

Precondition: The collection must not be empty.

Exception: `IEmptyException`

locate

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

Locates an element in the collection that is equal to the given element. Sets the cursor to point to the element in the collection, or invalidates the cursor if no such element exists.

If the collection contains several such elements, the first element in iteration order is located.

Precondition: The cursor must belong to the collection.

Return Value: Returns true if an element was found.

Exceptions: `ICursorInvalidException`

locateElementWithKey

`IBoolean locateElementWithKey (Key const& key, ICursor& cursor) const;`

Locates an element in the collection with the same key as the given key. Sets the cursor to point to the element in the collection, or invalidates the cursor if no such element exists.

If the collection contains several such elements, the first element in iteration order is located.

Precondition: The cursor must belong to the collection.

Return Value: Returns true if an element was found.

Exception: `ICursorInvalidException`

locateFirst

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

Locates the first element in iteration order in the collection that is equal to the given element. Sets the cursor to the located element, or invalidates the cursor if no such element exists.

Precondition: The cursor must belong to the collection.

Return Value: Returns true if an element was found.

Exception: `ICursorInvalidException`

locateLast

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

Locates the last element in iteration order in the collection that is equal to the given element. Sets the cursor to the located element, or invalidates the cursor if no such element exists.

Precondition: The cursor must belong to the collection.

Return Value: Returns true if an element was found.

Exception: `ICursorInvalidException`

locateNext

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

Locates the next element in iteration order in the collection that is equal to the given element, starting at the element next to the one pointed to by the given cursor. Sets the cursor to point to the element in the collection. The cursor is invalidated if the end of the collection is reached and no more occurrences of the given element are left to be visited.

Note: If you code a call to `locateFirst()` and a set of calls to `locateNext()`, each occurrence of an element will be visited exactly once in iteration order.

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

Return Value: Returns true if an element was found.

Exception: `ICursorInvalidException`

locateNextElementWithKey

```
IBoolean locateNextElementWithKey (
    Key const& key, ICursor& cursor ) const;
```

Locates the next element in iteration order in the collection with the given key, starting at the element next to the one pointed to by the given cursor. Sets the cursor to point to the element in the collection. The cursor is invalidated if the end of the collection is reached and no more occurrences of such an element are left to be visited.

Note: If you code a call to `locateFirst()` and a set of calls to `locateNextElementWithKey()`, each occurrence of an element will be visited exactly once in iteration order.

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

Return Value: Returns true if an element was found.

Exception: `ICursorInvalidException`

locateOrAdd

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

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

Locates an element in the collection that is equal to the given element. (See “locate” on page 112 for details on `locate()`.) If no such element is found, `locateOrAdd()` adds the element as described in “add” on page 98. 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 notifications send an `addId` notification.

Return Value: Returns `true` if the element was located. Returns `false` if the element could not be located but had to be added.

Exceptions

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

locateOrAddElementWithKey

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

```
IBoolean locateOrAddElementWithKey (  
    Element const& element; ICursor& cursor ) ;
```

Locates an element in the collection with the given key as described for the `locateElementWithKey()` function. If no such element exists, `locateOrAddElementWithKey()` adds the element as described in “add” on page 98. 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 notifications send an `addId` notification.

Return Value: Returns true if the element was located. Returns false if the element could not be located but had to be added.

Exceptions

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

locatePrevious

```
IBoolen 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`

notifier

```
INotifier const&
    notifier() const;
```

```
INotifier&
    notifier();
```

Returns a reference to the notifier object.

notifyObservers

```
INotifier&  
    notifyObservers( INotificationEvent const& );
```

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.

numberOfDifferentElements

```
INumber numberOfDifferentElements ( ) const;
```

Returns the number of different elements in the collection.

numberOfDifferentKeys

```
INumber numberOfDifferentKeys ( ) const;
```

Returns the number of different keys in the collection.

numberOfElements

```
INumber numberOfElements ( ) const;
```

Returns the number of elements the collection contains.

numberOfElementsWithKey

```
INumber numberOfElementsWithKey ( Key const& key ) const;
```

Returns the number of elements in the collection with the given key.

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 notifications send a removeId 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`

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 98), 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 notifications send an `addId` 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 119.

Side Effects

- If an element was removed, all cursors of this collection become undefined.
- If an element was removed, collection classes supporting notifications send a `removeId` notification.

Return Value: Returns true if an element was removed.

removeAll

```
INumber removeAll ( ) ;
```

Removes all elements from the collection. Element destructors are called as described in “removeAt” on page 119.

Side Effects

- All cursors of this collection become undefined.
- Collection classes supporting notifications send a modifyId 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 119.

Side Effects

- If any elements were removed, all cursors of this collection become undefined.
- If any elements were removed, collection classes supporting notifications send a modifyId 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 119.

Side Effects

- If any elements were removed, all cursors of this collection become undefined.
- If any elements were removed, collection classes supporting notifications send a removeId notification.

Return Value: The number of elements removed.

removeAllOccurrences

```
INumber removeAllOccurrences ( Element const& element ) ;
```

Removes all elements from the collection that are equal to the given element, and returns the number of elements removed. Element destructors are called as described in “removeAt” on page 119.

Side Effects

- If any elements were removed, all cursors of this collection become undefined.
- If any elements were removed, collection classes supporting notifications send a `modifyId` 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 notifications send a `removeId` 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, $(1 \leq position \leq \text{numberOfElements}())$.

Side Effects

- All cursors of this collection become undefined.
- Collection classes supporting notifications send a `removeId` notification.

Exception: `IPositionInvalidException`

removeElementWithKey

```
IBoolan 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`.”

Side Effects

- If an element was removed, all cursors of this collection become undefined.
- If an element was removed, collection classes supporting notifications send a `removeId` 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 119.

Precondition: The collection must not be empty.

Side Effects

- All cursors of this collection become undefined.
- If an element was removed, collection classes supporting notifications send a `removeId` notification.

Exception: `IEmptyException`

removeLast

```
void removeLast ( ) ;
```

Removes the last element from the collection. Element destructors are called as described in “`removeAt`” on page 119.

Precondition: The collection must not be empty.

Side Effects

- All cursors of this collection become undefined.
- If an element was removed, collection classes supporting notifications send a `removeId` 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 notifications send a `replaceId` notification.

Exceptions

- `ICursorInvalidException`
- `IInvalidReplacementException`

replaceElementWithKey

`IBoolean replaceElementWithKey (Element const& element) ;`

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

Replaces an element with the same key as the given element by the given element, and sets the cursor to this element. If no such element exists, it invalidates the cursor. In collections with nonunique elements, an arbitrary occurrence of such an element will be replaced.

Precondition: The cursor must belong to the collection.

Side Effect: Collection classes supporting notifications send a `replaceId` notification.

Return Value: Returns true if an element was replaced.

Exceptions: `ICursorInvalidException`

reverse

`void reverse () ;`

Reverses the sequence of elements in the collection.

Side Effects

- All cursors of this collection become undefined.
- Collection classes supporting notifications send a `modifyId` notification.

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`

setToNext

`IBoolean setToNext (ICursor& cursor) const;`

Sets the cursor to the next element in the collection in iteration order. If no more elements are left to be visited, the given cursor will no longer be valid.

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

Return Value: Returns true if there is a next element.

Exceptions: `ICursorInvalidException`

setToNextDifferentElement

`IBoolean setToNextDifferentElement (ICursor& cursor) const;`

Sets the cursor to the next element in iteration order in the collection that is different from the element pointed to by the given cursor. If no more elements are left to be visited, the given cursor will no longer be valid.

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

Return Value: Returns true if a subsequent element was found that is different.

Exception: `ICursorInvalidException`

setToNextWithDifferentKey

`IBoolean setToNextWithDifferentKey (ICursor& cursor) const;`

Sets the cursor to the next element in the collection in iteration order with a key different from the key of the element pointed to by the given cursor. If no such element exists, the given cursor is no longer valid.

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

Return Value: Returns true if a subsequent element was found whose key is different from the current key.

Exception: `ICursorInvalidException`

setToPosition

```
void setToPosition ( IPosition position, ICursor& cursor ) const;
```

Sets the cursor to the element at the given position. Position 1 specifies the first element.

Precondition

- The cursor must belong to the collection.
- Position must be a valid position in the collection; that is, $(1 \leq position \leq numberOfElements())$.

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 (element1 > element2)
0        if (element1 == element2)
<0      if (element1 < element2)
```

Side Effects

- All cursors of this collection become undefined.
- Collection classes supporting notifications send a `modifyId` notification.

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 notifications send a modifyId notification.

Exceptions

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

Chapter 16. 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 *IBM 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 $m > n$, and *zero* times if $m \leq n$.

Derivation

```
Collection
  Equality Collection
    Bag
```

Variants and Header Files

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

To use notifications with your collections, change the name of the desired collection class template in the list below from I... to IV....

Notification-enabled classes have the following additional members:

- “disableNotification” on page 108
- “enableNotification” on page 109
- “isEnabledForNotification” on page 111
- “notifier” on page 115
- “notifyObservers” on page 116

Class Name	Header File	Implementation Variant
IBag	ibag.h	List
IGBag	ibag.h	List
IBagAsHshTable	ibaghsh.h	Hash Table
IGBagAsHshTable	ibaghsh.h	Hash Table

Bag

Class Name	Header File	Implementation Variant
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 Chapter 14, “Introduction to Flat Collections” on page 93. The following members are provided for bag:

Method	Page	Method	Page
Constructor	97	isEmpty	110
Copy Constructor	97	isFull	111
Destructor	97	locate	112
operator!=	97	locateNext	113
operator=	98	locateOrAdd	113
operator==	98	maxNumberOfElements	115
add	98	newCursor	115
addAllFrom	99	numberOfDifferentElements	116
addDifference	102	numberOfElements	116
addIntersection	103	numberOfOccurrences	116
addUnion	104	remove	117
allElementsDo	105	removeAllOccurrences	118
anyElement	106	removeAll	118
contains	106	removeAt	119
containsAllFrom	106	replaceAt	120
copy	107	setToFirst	121
differenceWith	108	setToNext	122
elementAt	108	setToNextDifferentElement	122
intersectionWith	110	unionWith	124
isBounded	110		

Bag also defines a cursor that inherits from `IElementCursor`. The members for `IElementCursor` are described in Chapter 37, “Cursor” on page 257.

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 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
- 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 *IBM 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 Bag

The following program uses the default bag class, IBag, to create a bag. It fills the bag with the integers passed as arguments.

The program uses the the constant applicator class, IConstantApplicator, to summarize all integers with the `applyTo()` function. It uses the `add()` function to fill the bag and the `allElementsDo` function to summarize all elements of the bag.

```
/*-----*\
|  sumup.CPP - Sum up integers to demonstrate using iterators.
|              *****
|  Add all elements in a bag to demonstrate the use of
|  iterators with additional arguments.
|  The bag elements are integers passed as arguments to main().
|  \*-----*/

#include <iabag.h>
#include <iostream.h>
#include <stdlib.h>

typedef IBag <int> IntBag;

class SumApplicator : public IConstantApplicator <int> {
    int ivSum;
public:
    SumApplicator () : ivSum (0) {}
    IBoolean applyTo (int const& i) {
        ivSum += i;
        return True;
    }
    int sum () { return ivSum; }
};

int sumUsingApplicatorObject (IntBag const& bag) {
```

```

    SumApplicator sumUp;
    bag.allElementsDo (sumUp);
    return sumUp.sum ();
}

IBoolean sumUpFunction (int const& i, void* sum) {
    *(int*)sum += i;
    return True;
};

int sumUsingApplicatorFunction (IntBag const& bag) {
    int sum = 0;
    bag.allElementsDo (sumUpFunction, &sum);
    return sum;
}

int main (int argc, char* argv[]) {
    IntBag intbag;
    for (int cnt=1; cnt < argc; cnt++)
        intbag.add(atoi(argv[cnt]));

    cout << "Sum obtained using an Applicator Object = "
         << sumUsingApplicatorObject(intbag) << endl;

    cout << "Sum obtained using an Applicator Function = "
         << sumUsingApplicatorFunction(intbag) << endl;

    return 0;
}

```

The program produces the following output, using arguments 1 2 3 4 5

```

Sum obtained using an Applicator Object = 15
Sum obtained using an Applicator Function = 15

```


Chapter 17. 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 *IBM 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 notifications with your collections, change the name of the desired collection class template in the list below from I... to IV....

Notification-enabled classes have the following additional members:

- “disableNotification” on page 108
- “enableNotification” on page 109
- “isEnabledForNotification” on page 111
- “notifier” on page 115
- “notifyObservers” on page 116

Class Name	Header File	Implementation Variant
IDeque	idqu.h	List
IGDeque	idqu.h	List
IDequeAsList	idqulst.h	List
IGDequeAsList	idqulst.h	List
IDequeAsTable	idqutab.h	Table
IGDequeAsTable	idqutab.h	Table
IDequeAsDilTable	idqudil.h	Diluted table
IGDequeAsDilTable	idqudil.h	Diluted table

Members

All members of flat collections are described in Chapter 14, “Introduction to Flat Collections” on page 93. The following members are provided for deque:

Method	Page	Method	Page
Constructor	97	isFull	111
Copy Constructor	97	isLast	111
Destructor	97	lastElement	111
operator=	98	maxNumberOfElements	115
add	98	newCursor	115
addAllFrom	99	numberOfElements	116
addAsFirst	100	positionAt	117
addAsLast	100	removeAll	118
allElementsDo	105	removeFirst	120
anyElement	106	removeLast	120
compare	106	setToFirst	121
copy	107	setToLast	121
elementAt	108	setToNext	122
elementAtPosition	109	setToPosition	123
firstElement	110	setToPrevious	123
isBounded	110		
isEmpty	110		
isFirst	111		

Deque also defines a cursor that inherits from `IOrderedCursor`. The members for `IOrderedCursor` are described in Chapter 37, “Cursor” on page 257.

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 *IBM 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, `IDeque`, to create a deque. It fills the deque with characters by adding them to the back end. The program then removes the characters from alternating ends of the deque (beginning with the front end) until the deque is empty.

The program uses the constant iterator class, `IConstantIterator`, when printing the collection. It uses the `addAsLast()` function to fill the deque and the `numberOfElements()` function to determine the deque's size. It uses the functions `firstElement()`, `removeFirst()`, `lastElement()`, and `removeLast()` to empty the deque.

```

/*-----*\
| letterdq.CPP - Letter Double Ended Queue
|               This is an example of using a Deque.
|               *****
|-----*\

#include <iostream.h>

#include <idqu.h>
        // Let's use the default deque
typedef IDeque <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";

/*-----*\
| Main program
|-----*\
int main()
{
    Deque D;
    char C;
    IBoolean ReadFront = True;

    int i;

    // Put all characters in the deque.
    // Then read it, changing the end to read from
    // with every character read.

    cout << endl
        << "Adding characters to the back end of the deque:" << endl;

    for (i = 0; String[i] != 0; i++) {

```

```

        D.addAsLast(String[i]);
        cout << String[i];
    }

    cout << endl << endl
        << "Current number of elements in the deque: "
        << D.numberOfElements() << endl;

    cout << endl
        << "Contents of the deque:" << endl;
    Print Aprinter;
    D.allElementsDo(Aprinter);

    cout << endl << endl
        << "Reading from the deque one element from front, one "
        << "from back, and so on:" << endl;

    while (!D.isEmpty())
    {
        if (ReadFront)                // Read from front of Deque
        {
            C = D.firstElement();      // Get the character
            D.removeFirst();           // Delete it from the Deque
        }
        else
        {
            C = D.lastElement();
            D.removeLast();
        }
        cout << C;

        ReadFront = !ReadFront;      // Switch to other end of Deque
    }

    cout << endl;

    return(0);
}

```

The program produces the following output:

Adding characters to the back end of the deque:

Teqikbonfxjmsoe aydg.o zlarv pu o wr cu h

Current number of elements in the deque: 42

Contents of the deque:

Teqikbonfxjmsoe aydg.o zlarv pu 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.

Chapter 18. 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 *IBM 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 notifications with your collections, change the name of the desired collection class template in the list below from `I...` to `IV...`.

Notification-enabled classes have the following additional members:

- “disableNotification” on page 108
- “enableNotification” on page 109
- “isEnabledForNotification” on page 111
- “notifier” on page 115
- “notifyObservers” on page 116

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 Chapter 14, “Introduction to Flat Collections” on page 93. The following members are provided for equality sequence:

Method	Page	Method	Page
Constructor	97	lastElement	111
Copy Constructor	97	locate	112
Destructor	97	locateFirst	112
operator!=	97	locateLast	112
operator=	98	locateNext	113
operator==	98	locateOrAdd	113
add	98	locatePrevious	115
addAllFrom	99	maxNumberOfElements	115
addAsFirst	100	newCursor	115
addAsLast	100	numberOfElements	116
addAsNext	101	numberOfOccurrences	116
addAsPrevious	101	positionAt	117
addAtPosition	101	remove	117
allElementsDo	105	removeAll	118
anyElement	106	removeAllOccurrences	118
compare	106	removeAt	119
contains	106	removeAtPosition	119
containsAllFrom	106	removeFirst	120
copy	107	removeLast	120
elementAt	108	replaceAt	120
elementAtPosition	109	reverse	121
firstElement	110	setToFirst	121
isBounded	110	setToLast	121
isEmpty	110	setToNext	122
isFirst	111	setToPosition	123
isFull	111	setToPrevious	123
isLast	111	sort	123

Equality sequence also defines a cursor that inherits from `IOrderedCursor`. The members for `IOrderedCursor` are described in Chapter 37, “Cursor” on page 257.

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:

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

Equality Sequence

- 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 *IBM Open Class Library User's Guide* for more information.

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

Chapter 19. 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 *IBM 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 notifications with your collections, change the name of the desired collection class template in the list below from I... to IV....

Notification-enabled classes have the following additional members:

- “disableNotification” on page 108
- “enableNotification” on page 109
- “isEnabledForNotification” on page 111
- “notifier” on page 115
- “notifyObservers” on page 116

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 Chapter 14, “Introduction to Flat Collections” on page 93. The following members are provided for heap:

Method	Page	Method	Page
Constructor	97	replaceAt	120
Copy Constructor	97	setToFirst	121
Destructor	97	setToNext	122
operator=	98		
add	98		
addAllFrom	99		
allElementsDo	105		
anyElement	106		
copy	107		
elementAt	108		
isBounded	110		
isEmpty	110		
isFull	111		
maxNumberOfElements	115		
newCursor	115		
numberOfElements	116		
removeAll	118		
removeAt	119		

Heap also defines a cursor that inherits from `IElementCursor`. The members for `IElementCursor` are described in Chapter 37, “Cursor” on page 257.

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 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 *IBM 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 167 for an example of using a heap.

Chapter 20. 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 *IBM 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 *IBM 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 notifications with your collections, change the name of the desired collection class template in the list below from I... to IV....

Notification-enabled classes have the following additional members:

- "disableNotification" on page 108
- "enableNotification" on page 109
- "isEnabledForNotification" on page 111
- "notifier" on page 115
- "notifyObservers" on page 116

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

Key Bag

Members

All members of flat collections are described in Chapter 14, “Introduction to Flat Collections” on page 93. The following members are provided for key bag:

Method	Page	Method	Page
Constructor	97	locateElementWithKey	112
Copy Constructor	97	locateNextElementWithKey	113
Destructor	97	locateOrAddElementWithKey	114
operator=	98	maxNumberOfElements	115
add	98	newCursor	115
addAllFrom	99	numberOfDifferentKeys	116
addOrReplaceElementWithKey	103	numberOfElements	116
allElementsDo	105	numberOfElementsWithKey	116
anyElement	106	removeAll	118
containsAllKeysFrom	107	removeAllElementsWithKey	118
containsElementWithKey	107	removeAt	119
copy	107	removeElementWithKey	119
elementAt	108	replaceAt	120
elementWithKey	109	replaceElementWithKey	121
isBounded	110	setToFirst	121
isEmpty	110	setToNext	122
isFull	111	setToNextWithDifferentKey	122
key	111		

Key Bag also defines a cursor that inherits from `IElementCursor`. The members for `IElementCursor` are described in Chapter 37, “Cursor” on page 257.

Template Arguments and Required Functions

Key Bag

`IKeyBag` <*Element*, *Key*>
`IGKeyBag` <*Element*, *Key*, *KEHOps*>

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

Element Type

- Copy constructor
- Destructor
- Assignment
- Key access

Key Type

- Equality test
- Hash function

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 *IBM 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 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 587 for the code of the animal.h file.

Key Bag

```
/*-----*/
| animals.CPP - Example for the use of the Key Bag          |
|               *****                                   |
| We keep a Key Bag of our observations on animals. Elements |
| handled in this Key Bag are of type animal, the key is the  |
| name of the animal.                                         |
| This Key Bag allows us to efficiently access all observations |
| on an animal.                                               |
| We use a Sequence to store the names of all extinct animals. |
| At last we remove all extinct animals from the Key Bag.     |
|-----*/

#include <iostream.h>
        // Class Animal:
#include "animal.h"

        // Let's use the default Key Bag:
#include <ikb.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();

    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);
    } while (animalsCur1.hasMoreElements());
}
```

```

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.numberElements() << endl;

cout << endl
    << "Number of different animals: "
    << animals.numberDifferentKeys() << endl;

return 0;
}

```

The program produces the following output:

```

All our observations on animals:
The mammoth is extinct.
The mammoth is long haired.
The dinosaur is heavy.
The dinosaur is huge.
The dinosaur is extinct.
The sabre tooth tiger is extinct.
The zebra is striped.
The bear is strong.
The bear is heavy.
The lion is strong.
The lion is dangerous.
The eagle is black.
The eagle is strong.

```

```

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:

```

Key Bag

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

Chapter 21. 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 *IBM 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 *IBM 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 notifications with your collections, change the name of the desired collection class template in the list below from `I...` to `IV...`.

Notification-enabled classes have the following additional members:

- “disableNotification” on page 108
- “enableNotification” on page 109
- “isEnabledForNotification” on page 111
- “notifier” on page 115
- “notifyObservers” on page 116

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
<code>IKeySetAsHshTable</code>	<code>ikshsh.h</code>	Hash table

Key Set

Class Name	Header File	Implementation Variant
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 Chapter 14, “Introduction to Flat Collections” on page 93. The following members are provided for key set:

Method	Page	Method	Page
Constructor	97	isEmpty	110
Copy Constructor	97	isFull	111
Destructor	97	key	111
operator=	98	locateElementWithKey	112
add	98	locateOrAddElementWithKey	114
addAllFrom	99	maxNumberOfElements	115
addOrReplaceElementWithKey	103	newCursor	115
allElementsDo	105	numberOfElements	116
anyElement	106	removeAll	118
containsAllKeysFrom	107	removeAt	119
containsElementWithKey	107	removeElementWithKey	119
copy	107	replaceAt	120
elementAt	108	replaceElementWithKey	121
elementWithKey	109	setToFirst	121
isBounded	110	setToNext	122

Key set also defines a cursor that inherits from `IElementCursor`. The members for `IElementCursor` are described in Chapter 37, “Cursor” on page 257.

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:

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:

Key Set

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:

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

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 *IBM 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 587 for the code of the demoelem.h file.

```
/*-----*\
| intkysset.CPP - Integer Key Set for demonstration of using
|               a KeySet.
\*-----*/
#include <iostream.h>
#include <iglobals.h>

#include <iks.h>
// Class DemoElement:
#include "demoelem.h"

#ifdef IC_PAGETUNE
#define _INTKYSET_CPP_
#include <ipagetun.h>
#endif

typedef IKeySet < DemoElement, int > TestKeySet;

ostream & operator << ( ostream & sout, TestKeySet const & t)
{ sout << t.numberOfElements() << " elements are in the set:" << endl;

  TestKeySet::Cursor cursor (t);

  // forICursor(c)
```

Key Set

```
// expands to
// for ((c).setToFirst (); (c).isValid (); (c).setToNext ())

forICursor (cursor)
    sout << "    " << cursor.element () << endl;

return sout << endl;
}

main()
{
    TestKeySet t;

    t.add(DemoElement(1,1));
    t.add(DemoElement(2,4711));
    t.add(DemoElement(3,1));
    t.add(DemoElement(4,443));

    cout << t;

    t.removeElementWithKey (3);

    cout << t;

    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
```

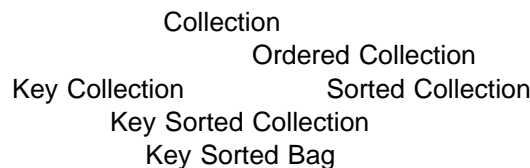
Chapter 22. Key Sorted Bag

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 *IBM 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 notifications with your collections, change the name of the desired collection class template in the list below from I... to IV....

Notification-enabled classes have the following additional members:

- “disableNotification” on page 108
- “enableNotification” on page 109
- “isEnabledForNotification” on page 111
- “notifier” on page 115
- “notifyObservers” on page 116

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

Members

All members of flat collections are described in Chapter 14, “Introduction to Flat Collections” on page 93. The following members are provided for key sorted bag:

Method	Page	Method	Page
Constructor	97	numberOfDifferentKeys	116
Copy Constructor	97	numberOfElements	116
Destructor	97	numberOfElementsWithKey	116
operator=	98	positionAt	117
add	98	removeAll	118
addAllFrom	99	removeAllElementsWithKey	118
addOrReplaceElementWithKey	103	removeAt	119
allElementsDo	105	removeAtPosition	119
anyElement	106	removeElementWithKey	119
compare	106	removeFirst	120
containsAllKeysFrom	107	removeLast	120
containsElementWithKey	107	replaceAt	120
copy	107	replaceElementWithKey	121
elementAt	108	setToFirst	121
elementAtPosition	109	setToLast	121
elementWithKey	109	setToNext	122
firstElement	110	setToNextWithDifferentKey	122
isBounded	110	setToPosition	123
isEmpty	110	setToPrevious	123
isFirst	111		
isFull	111		
isLast	111		
key	111		
lastElement	111		
locateElementWithKey	112		
locateNextElementWithKey	113		
locateOrAddElementWithKey	114		
maxNumberOfElements	115		
newCursor	115		

Key sorted bag also defines a cursor that inherits from `IOrderedCursor`. The members for `IOrderedCursor` are described in Chapter 37, “Cursor” on page 257.

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>*

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

`IKeySortedBag<Element,Key>`

For polymorphism, you can use the corresponding abstract class, `IKeySortedBag`, which is found in the `iaksb.h` header file. See the section on *Polymorphism and the Collections* in the *IBM 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 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 587 for the code of the `toyword.h` file.

```
// WORDBAG - An example of using a Key Sorted Bag
#include <iostream.h>

// Class Word
#include "toyword.h"

// Let's use the defaults:
#include <iaksb.h>

typedef IKeySortedBag <Word, unsigned> WordBag;

int main()
{
    IString phrase[] = {"people", "who", "live", "in", "glass",
                       "houses", "should", "not", "throw", "stones"};
    const size_t phraseWords = sizeof(phrase) / sizeof(IString);

    WordBag wordbag(phraseWords);

    for (int cnt=0; cnt < phraseWords; cnt++) {
        unsigned keyValue = phrase[cnt].length();
        Word theWord(phrase[cnt],keyValue);
```

```

    wordbag.add (theWord);
}

cout << "Contents of our WordBag sorted by number of letters:" << endl;

WordBag::Cursor wordBagCursor(wordbag);
forCursor(wordBagCursor)
    cout << "WB: " << wordBagCursor.element().getWord() << endl;

cout << endl << "Our phrase has " << phraseWords << " words." << endl;
cout << "In our WordBag are " << wordbag.numberOfElements()
    << " words." << endl << endl;

cout << "There are " << wordbag.numberOfDifferentKeys()
    << " different word lengths." << endl << endl;

wordBagCursor.setToFirst();
do {
    unsigned letters = wordbag.key(wordBagCursor.element());
    cout << "There are "
        << wordbag.numberOfElementsWithKey(letters)
        << " words with " << letters << " letters." << endl;
} while (wordbag.setToNextWithDifferentKey(wordBagCursor));

return 0;
}

```

This program produces the following output:

```

Contents of our WordBag sorted by number of letters:
WB: in
WB: who
WB: not
WB: live
WB: glass
WB: throw
WB: people
WB: houses
WB: should
WB: stones

```

```

Our phrase has 10 words.
In our WordBag are 10 words.

```

```

There are 5 different word lengths.

```

```

There are 1 words with 2 letters.
There are 2 words with 3 letters.
There are 1 words with 4 letters.
There are 2 words with 5 letters.
There are 4 words with 6 letters.

```


Chapter 23. 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 *IBM 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

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

To use notifications with your collections, change the name of the desired collection class template in the list below from I... to IV....

Notification-enabled classes have the following additional members:

- "disableNotification" on page 108
- "enableNotification" on page 109
- "isEnabledForNotification" on page 111
- "notifier" on page 115
- "notifyObservers" on page 116

Class Name	Header File	Implementation Variant
IKKeySortedSet	ikss.h	AVL tree
IGKeySortedSet	ikss.h	AVL tree
IKKeySortedSetAsAvlTree	ikssavl.h	AVL tree
IGKeySortedSetAsAvlTree	ikssavl.h	AVL tree
IKKeySortedSetAsBstTree	ikssbst.h	B* tree
IGKeySortedSetAsBstTree	ikssbst.h	B* tree

Key Sorted Set

Class Name	Header File	Implementation Variant
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 Chapter 14, “Introduction to Flat Collections” on page 93. The following members are provided for key sorted set:

Method	Page	Method	Page
Constructor	97	key	111
Copy Constructor	97	lastElement	111
Destructor	97	locateElementWithKey	112
operator=	98	locateNextElementWithKey	113
add	98	locateOrAddElementWithKey	114
addAllFrom	99	maxNumberOfElements	115
addOrReplaceElementWithKey	103	newCursor	115
allElementsDo	105	numberOfElements	116
anyElement	106	positionAt	117
compare	106	removeAll	118
containsAllKeysFrom	107	removeAt	119
containsElementWithKey	107	removeAtPosition	119
copy	107	removeElementWithKey	119
elementAt	108	removeFirst	120
elementAtPosition	109	removeLast	120
elementWithKey	109	replaceAt	120
firstElement	110	replaceElementWithKey	121
isBounded	110	setToFirst	121
isEmpty	110	setToLast	121
isFirst	111	setToNext	122
isFull	111	setToPosition	123
isLast	111	setToPrevious	123

Key Sorted Set also defines a cursor that inherits from `IOrderedCursor`. The members for `IOrderedCursor` are described in Chapter 37, “Cursor” on page 257.

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>
IGKeySortedSetAsBstTree <Element, Key, KCOps>
```

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

Element Type

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

Key Type

Ordering relation

Key Sorted Set 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

Key Sorted Set

- 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 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 *IBM 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.

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 587 for the code of the `parcel.h` file.

```

/*-----*/
parcel.CPP - Parcels are handled using a KeySorted Set and
              a Heap.
              *****

We maintain two collections that keep track of parcels in
circulation and parcels delivered. The collection for the
parcels in circulation is a KeySorted Set (key, sorted,
unique elements, no element equality). For the delivered
parcels we do not care about fast or sorted retrieval.
So we select the Heap for this collection (no key, unordered,
multiple elements, no element equality).

A parcel has as member data two objects of type PlaceTime,
which is a point in space and time: one object for its origin,
one for its current place and time. It also has as member
data two objects of type IString, for the destination and
for the ID.

Function updateParcels adds parcels that have arrived at
their destination to the collection for delivered parcels,
and removes them from the collection of circulating parcels.
This demonstrates the use of removeAll().

/*-----*/
#include <iostream.h>

#include "parcel.h"
// Let's use the default KeySorted Set:
#include <ikss.h>
// Let's use the default Heap:
#include <ihp.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() {

```

Key Sorted Set

```
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 << endl << "The situation at start:" << endl;
cout << "Parcels in circulation:" << endl << circulating;

today ++;
circulating.elementAtWithKey("27AmTo").arrivedAt(
    "Atlanta", today);
circulating.elementAtWithKey("25WaSt").arrivedAt(
    "Amsterdam", today);
circulating.elementAtWithKey("25DuKa").arrivedAt(
    "Paris", today);
update(circulating, delivered);
cout << endl << endl << "The situation at day " << today << ":"
    << endl;
cout << "Parcels in circulation:" << endl << 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 << endl << endl << "The situation at day " << today << ":"
    << endl;
cout << "Parcels in circulation:" << endl << circulating;

today ++;
circulating.elementAtWithKey("25WaSt").arrivedAt("Oslo", today);
circulating.elementAtWithKey("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 << endl << endl << "The situation at day " << today << ":"
    << endl;
cout << "Parcels in circulation:" << endl << circulating;
cout << "Parcels delivered:" << endl << delivered;

    // Now we make all parcels arrive today:
today ++;

ParcelSet::Cursor circulatingcursor(circulating);
forICursor(circulatingcursor) {
    circulating.elementAt(circulatingcursor).wasDelivered(today);
}
update(circulating, delivered);
cout << endl << endl << "The situation at day " << today << ":"
    << endl;
cout << "Parcels in circulation:" << endl << circulating;
cout << "Parcels delivered:" << endl << delivered;

if (circulating.isEmpty())
    cout << endl << "All parcels were delivered." << endl;
else
    cout << endl << "Something very strange happened here." << endl;
```

```

    return 0;
}

ostream& operator<<(ostream& os, ParcelSet const& parcels) {
    ParcelSet::Cursor pcursor(parcels);
    forICursor(pcursor) {
        os << pcursor.element() << endl;
    }
    return os;
}

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

IBoolen wasDelivered(Parcel const& p, void* dp) {
    if ( p.lastArrival().city() == p.destination() ) {
        ((ParcelHeap*)dp)->add(p);
        return True;
    }
    else
        return False;
}

void update(ParcelSet& p, ParcelHeap& d) {
    p.removeAll(wasDelivered, &d);
}

```

The program produces the following output:

```

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

```

```

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

```

```

The situation at day 18:
Parcels in circulation:
25DuKa: From Dublin(day 16) to Kairo
         is at Paris since day 17.
25WaSt: From Washington(day 15) to Stockholm
         is at Amsterdam since day 17.
27AmTo: From Amsterdam(day 10) to Toronto
         is at Chicago since day 18.

```

Key Sorted Set

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.

Chapter 24. 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.

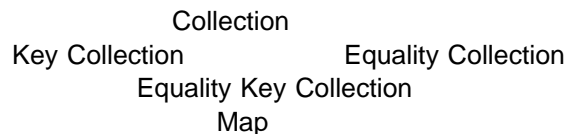
Only elements with unique keys are supported. A request to add an element whose key already exists in another element of the collection causes an exception to be thrown. A request to add a duplicate element is ignored.

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 *IBM 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 *IBM 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 notifications with your collections, change the name of the desired collection class template in the list below from I... to IV....

Notification-enabled classes have the following additional members:

- "disableNotification" on page 108
- "enableNotification" on page 109
- "isEnabledForNotification" on page 111
- "notifier" on page 115
- "notifyObservers" on page 116

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

Map

Class Name	Header File	Implementation Variant
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 Chapter 14, “Introduction to Flat Collections” on page 93. The following members are provided for map:

Method	Page	Method	Page
Constructor	97	intersectionWith	110
Copy Constructor	97	isBounded	110
Destructor	97	isEmpty	110
operator!=	97	isFull	111
operator=	98	key	111
operator==	98	locate	112
add	98	locateElementWithKey	112
addAllFrom	99	locateOrAdd	113
addDifference	102	locateOrAddElementWithKey	114
addIntersection	103	maxNumberOfElements	115
addOrReplaceElementWithKey	103	newCursor	115
addUnion	104	numberOfElements	116
allElementsDo	105	remove	117
anyElement	106	removeAll	118
contains	106	removeAt	119
containsAllFrom	106	removeElementWithKey	119
containsAllKeysFrom	107	replaceAt	120
containsElementWithKey	107	replaceElementWithKey	121
copy	107	setToFirst	121
differenceWith	108	setToNext	122
elementAt	108	unionWith	124
elementWithKey	109		

Map also defines a cursor that inherits from IElementCursor. The members for IElementCursor are described in Chapter 37, “Cursor” on page 257.

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
- 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

Map

- 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 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

```
IMapAsDi1Table <Element, Key>  
IGMapAsDi1Table <Element, Key, EKCOps>
```

The implementation of the class `IMapAsDi1Table` 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, EKEH0ps>

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

Element Type

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

Key Type

- Equality test
- Hash function

Abstract Class

IMap<Element, Key>

For polymorphism, you can use the corresponding abstract class, IMap, which is found in the `imap.h` header file. See the section on *Polymorphism and the Collections* in the *IBM 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 587 for the code of the files `transelm.h`, `trmapops.h`, and `xebc2asc.h`.

```
/*-----*\
| transtab.CPP - Translation table to demonstrate using a Map
|               ""
|
| This example demonstrates the use of a Map through a
| bidirectional mapping between the 256 EBCDIC characters and
| the 256 ASCII code points.
|
| We build a map with 256 elements, each of which has
| an ebcCode and an ascCode.
| For EBCDIC-ASCII translation we want key access with
| ebcCode as key,
| for ASCII-EBCDIC translation we want key access with
| ascCode as key.
| Therefore this example demonstrates the principle of using
| different keys on the same element type when stored in
| different collections.
|
| What you can learn from this example:
| -----
| - What to do to use the map abstraction.
| - How to specify the required element functions in two ways:
|   1. defining operators for the element to store in the map
|   2. defining an operation class that contains
|      element and key functions
| - How to use the same element with different keys in
|   different maps.
|
| This example does not show the most efficient way of
| implementing an ASCII-EBCDIC translation.
|
|-----*\
#include "transelm.h"

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

#include "trmapops.h"

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

/*-----*\
| Now we define the two Map templates and two maps.
| We want both of them to be based on the Hashtable KeySet.
|-----*\
#include <imaphsh.h>

typedef IMapAsHshTable
    < TranslationElement, char, TranslationOpsE2A > TransE2AMap;
```

```

typedef IMapAsHshTable
    < TranslationElement, char, TranslationOpsA2E > TransA2EMap;

void display(char*, char*);

int main(int argc, char* argv[]) {

    TransA2EMap  A2EMap;
    TransE2AMap  E2AMap;

    /*-----*\
    | Load the translation table into both maps.      |
    | The maps organize themselves according to the key |
    | specification already given.                      |
    \*-----*/
    for (int i=0; i < 256; i++)
    {
        /*      ascCode      ebcCode      */
        TranslationElement te(translationTable[i], i);

        E2AMap.add(te);
        A2EMap.add(te);
    }

    // What do we want to convert now?
    char* toConvert;
    if (argc > 1) toConvert = argv[1];
    else          toConvert = "$7 (=Dollar seven)";

    size_t textLength = strlen(toConvert) + 1;

    char* convertedToAsc = new char[textLength];
    char* convertedToEbc = new char[textLength];

    // Convert the strings in place, character by character
    for (i=0; toConvert[i] != 0x00; i++) {
        convertedToAsc[i]
            = E2AMap.elementWithKey(toConvert[i]).ascCode ();
        convertedToEbc[i]
            = A2EMap.elementWithKey(toConvert[i]).ebcCode ();
    }

    display("To convert", toConvert);
    display("After EBCDIC-ASCII conversion", convertedToAsc);
    display("After ASCII-EBCDIC conversion", convertedToEbc);

    delete[] convertedToAsc;
    delete[] convertedToEbc;

    return 0;
}

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

void display (char* title, char* text) {
    cout << endl << title << ':' << endl;
    cout << " Text: '" << text << "'" << endl;
    cout << " Hex: " << hex;
    for (int i=0; text[i] != 0x00; i++) {
        cout << (int)(unsigned char) text[i] << " ";
    }
    cout << dec << endl;
}

```

The program produces the following output:

To convert:

Text: '\$7 (=Dollar seven)'

Hex: 5b f7 40 40 4d 7e c4 96 93 93 81 99 40 a2 85 a5 85 95 5d

After EBCDIC-ASCII conversion:

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

After ASCII-EBCDIC conversion:

Hex: ad fb 7c 7c d4 a1 bf 8a 9d 9d 20 56 7c 2c eb 55 eb 2b bd

Chapter 25. 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 *IBM Open Class Library User's Guide* for a complete illustration. See "Restricted Access" in the *IBM 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 notifications with your collections, change the name of the desired collection class template in the list below from I... to IV....

Notification-enabled classes have the following additional members:

- "disableNotification" on page 108
- "enableNotification" on page 109
- "isEnabledForNotification" on page 111
- "notifier" on page 115
- "notifyObservers" on page 116

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 Chapter 14, “Introduction to Flat Collections” on page 93. The following members are provided for priority queue:

Method	Page	Method	Page
Constructor	97	isFull	111
Copy Constructor	97	isLast	111
Destructor	97	key	111
operator=	98	lastElement	111
add	98	locateElementWithKey	112
addAllFrom	99	locateNextElementWithKey	113
allElementsDo	105	locateOrAddElementWithKey	114
anyElement	106	maxNumberOfElements	115
compare	106	newCursor	115
containsAllKeysFrom	107	numberOfDifferentKeys	116
containsElementWithKey	107	numberOfElements	116
copy	107	numberOfElementsWithKey	116
dequeue	108	positionAt	117
elementAt	108	removeAll	118
elementAtPosition	109	removeFirst	120
elementWithKey	109	setToFirst	121
enqueue	109	setToLast	121
firstElement	110	setToNext	122
isBounded	110	setToNextWithDifferentKey	122
isEmpty	110	setToPosition	123
isFirst	111	setToPrevious	123

Priority queue also defines a cursor that inherits from `IOrderedCursor`. The members for `IOrderedCursor` are described in Chapter 37, “Cursor” on page 257.

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:

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

```
IPriorityQueue<Element,Key>
```

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

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

Chapter 26. 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 *IBM 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 notifications with your collections, change the name of the desired collection class template in the list below from I... to IV....

Notification-enabled classes have the following additional members:

- "disableNotification" on page 108
- "enableNotification" on page 109
- "isEnabledForNotification" on page 111
- "notifier" on page 115
- "notifyObservers" on page 116

Queue

Class Name	Header File	Implementation Variant
IQueue	iqu.h	List
IGQueue	iqu.h	List
IQueueAsList	iqulst.h	List
IGQueueAsList	iqulst.h	List
IQueueAsTable	iqutab.h	Table
IGQueueAsTable	iqutab.h	Table
IQueueAsDilTable	iqudil.h	Diluted table
IGQueueAsDilTable	iqudil.h	Diluted table

Members

All members of flat collections are described in Chapter 14, “Introduction to Flat Collections” on page 93. The following members are provided for queue:

Method	Page	Method	Page
Constructor	97	isBounded	110
Copy Constructor	97	isEmpty	110
Destructor	97	isFirst	111
operator=	98	isFull	111
add	98	isLast	111
addAllFrom	99	lastElement	111
addAsLast	100	maxNumberOfElements	115
allElementsDo	105	newCursor	115
anyElement	106	numberOfElements	116
compare	106	positionAt	117
copy	107	removeAll	118
dequeue	108	removeFirst	120
elementAt	108	setToFirst	121
elementAtPosition	109	setToLast	121
enqueue	109	setToNext	122
firstElement	110	setToPosition	123

Queue also defines a cursor that inherits from `IOrderedCursor`. The members for `IOrderedCursor` are described in Chapter 37, “Cursor” on page 257.

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>
```

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 *IBM Open Class Library User's Guide* for further information.

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

Chapter 27. 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.

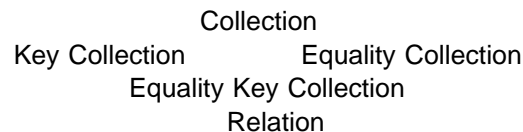
The keys of the elements are not unique. You can add an element whether or not there is already an element in the collection with the same key.

The figure “Behavior of add for Unique and Multiple Collections” in the *IBM 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 *IBM 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 notifications with your collections, change the name of the desired collection class template from `I...` to `IV...` (change `IRelation` to `IVRelation`, and `IGRelation` to `IVGRelation`).

Notification-enabled classes have the following additional members:

- “disableNotification” on page 108
- “enableNotification” on page 109
- “isEnabledForNotification” on page 111
- “notifier” on page 115
- “notifyObservers” on page 116

Relation

Members

All members of flat collections are described in Chapter 14, “Introduction to Flat Collections” on page 93. The following members are provided for relation:

Method	Page	Method	Page
Constructor	97	key	111
Copy Constructor	97	locate	112
Destructor	97	locateElementWithKey	112
operator!=	97	locateNextElementWithKey	113
operator=	98	locateOrAdd	113
operator==	98	locateOrAddElementWithKey	114
add	98	maxNumberOfElements	115
addAllFrom	99	newCursor	115
addDifference	102	numberOfDifferentKeys	116
addIntersection	103	numberOfElements	116
addOrReplaceElementWithKey	103	numberOfElementsWithKey	116
addUnion	104	remove	117
allElementsDo	105	removeAll	118
anyElement	106	removeAllElementsWithKey	118
contains	106	removeAt	119
containsAllFrom	106	removeElementWithKey	119
containsAllKeysFrom	107	replaceAt	120
containsElementWithKey	107	replaceElementWithKey	121
copy	107	setToFirst	121
differenceWith	108	setToNext	122
elementAt	108	setToNextWithDifferentKey	122
elementWithKey	109	unionWith	124
intersectionWith	110		
isBounded	110		
isEmpty	110		
isFull	111		

Relation also defines a cursor that inherits from `IElementCursor`. The members for `IElementCursor` are described in Chapter 37, “Cursor” on page 257.

Template Arguments and Required Functions

`IRelation` <*Element*, *Key*>
`IGRelation` <*Element*, *Key*, *EKEH0ps*>

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

Element Type

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

Key Type

- Equality test
- Hash function

Abstract Class

`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 *IBM Open Class Library User's Guide* for further information.

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

Chapter 28. 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 Chapter 18, “Equality Sequence” on page 137 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 *IBM 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

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

To use notifications with your collections, change the name of the desired collection class template in the list below from I... to IV....

Notification-enabled classes have the following additional members:

- “disableNotification” on page 108
- “enableNotification” on page 109
- “isEnabledForNotification” on page 111
- “notifier” on page 115
- “notifyObservers” on page 116

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 Chapter 14, “Introduction to Flat Collections” on page 93. The following members are provided for sequence:

Method	Page	Method	Page
Constructor	97	isLast	111
Copy Constructor	97	lastElement	111
Destructor	97	maxNumberOfElements	115
operator=	98	newCursor	115
add	98	numberOfElements	116
addAllFrom	99	positionAt	117
addAsFirst	100	removeAll	118
addAsLast	100	removeAt	119
addAsNext	101	removeAtPosition	119
addAsPrevious	101	removeFirst	120
addAtPosition	101	removeLast	120
allElementsDo	105	replaceAt	120
anyElement	106	replaceAt	120
compare	106	reverse	121
copy	107	setToLast	121
elementAt	108	setToNext	122
elementAtPosition	109	setToPosition	123
firstElement	110	setToPrevious	123
isBounded	110	sort	123
isEmpty	110		
isFirst	111		
isFull	111		

Sequence also defines a cursor that inherits from `IOrderedCursor`. The members for `IOrderedCursor` are described in Chapter 37, “Cursor” on page 257.

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 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

```
IASequence<Element>
```

For polymorphism, you can use the corresponding abstract class, `IASequence`, which is found in the `iaseq.h` header file. See the section on *Polymorphism and the Collections* in the *IBM 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 587 for the code of the `toyword.h` file.

```
/*-----*\
| wordseq.CPP - Example for using the Sequence.
|               *****
| This Sequence is used to handle elements of type Word.
|
| This example also demonstrates two different ways of
| iteration, using an object of an applicator class
| and using a cursor.
|-----*\

#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> WordApplicator;

/*-----*\
| Our Applicator class for use with allElementsDo().
|
| The alternative method of iteration, using a cursor, does
| not need such an applicator class. If you want to see
| this alternative, search for occurrences of cursor below.
|               *****
|-----*\
class PrintClass : public WordApplicator
{
public:
    IBoolean applyTo(Word &w)
    {
        cout << endl << w.getWord();    // Print the string
        return(True);
    }
};
```



```

/*-----*\
| Main program                                     |
\*-----*/
int main() {

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

    WordSeq WL;
    WordSeq::Cursor cursor(WL);
    PrintClass Print;

    int i;

    for (i = 0; i < 9; i++) {    // Put all strings into Sequence
        Word aWord(wordArray[i]); // Fill object with right value
        WL.addAsLast(aWord);     // Add it to the Sequence at end
    }

    cout << endl << "Sequence in initial order:" << endl;
    WL.allElementsDo(Print);

    WL.sort(wordCompare);        // Sort the Sequence ascending
    cout << endl << endl << "Sequence in sorted order:" << endl;
    WL.allElementsDo(Print);

    // Use iteration via cursor now:

    cout << endl << endl << "Look for \"fox\" in the Sequence:" << endl;
    for (cursor.setToFirst();
        cursor.isValid() && (WL.elementAt(cursor).getWord() != "fox");
        cursor.setToNext());

    if (WL.elementAt(cursor).getWord() != "fox") {
        cout << endl << "The element was not found." << endl;
    }
    else {
        cout << endl << " The element was found." << endl;
    }

    cout << endl << "The element at position 9: "
        << WL.elementAtPosition(9).getWord()
        << endl;

    return(0);
}

```

Sequence

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

Chapter 29. 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 *IBM 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 notifications with your collections, change the name of the desired collection class template in the list below from I... to IV....

Notification-enabled classes have the following additional members:

- “disableNotification” on page 108
- “enableNotification” on page 109
- “isEnabledForNotification” on page 111
- “notifier” on page 115
- “notifyObservers” on page 116

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 Chapter 14, “Introduction to Flat Collections” on page 93. The following members are provided for set:

Method	Page	Method	Page
Constructor	97	elementAt	108
Copy Constructor	97	intersectionWith	110
Destructor	97	isBounded	110
operator!=	97	isEmpty	110
operator=	98	isFull	111
operator==	98	locate	112
add	98	locateOrAdd	113
addAllFrom	99	maxNumberOfElements	115
addDifference	102	newCursor	115
addIntersection	103	numberOfElements	116
addUnion	104	remove	117
allElementsDo	105	removeAll	118
anyElement	106	removeAt	119
contains	106	replaceAt	120
containsAllFrom	106	setToFirst	121
copy	107	setToNext	122
differenceWith	108	unionWith	124

Set also defines a cursor that inherits from `IElementCursor`. The members for `IElementCursor` are described in Chapter 37, “Cursor” on page 257.

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 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

`ISet<Element>`

For polymorphism, you can use the corresponding abstract class, `ISet`, which is found in the `iset.h` header file. See the section on *Polymorphism and the Collections* in the *IBM 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 following 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.CPP - Even and Odd numbers are handled in different
|               Sets do demonstrate using Sets.
|               """"
|-----*/

#include <iostream.h>

#include <iset.h>      // Take the defaults for the Set and for
                      // the required functions for integer
typedef ISet <int> IntSet;

/*-----*\
| For iteration we want to use an object of an iterator class
|-----*/
class PrintClass : public IIterator<int> {
public:
    virtual IBoolean applyTo(int& i)
    { cout << " " << i << " "; return True;}
};

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

/*-----*\
| Main program
|-----*/
int main () {
    IntSet odd, prime;
    IntSet oddPrime, evenPrime;

    int One = 1, Two = 2, Three = 3, Five = 5, Seven = 7, Nine = 9;

    // Fill odd set with odd integers < 10
    odd.add( One );
    odd.add( Three );
```

Set

```
        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
```

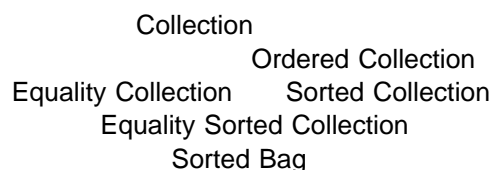

Chapter 30. Sorted Bag

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 *IBM 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 notifications with your collections, change the name of the desired collection class template in the list below from I... to IV....

Notification-enabled classes have the following additional members:

- “disableNotification” on page 108
- “enableNotification” on page 109
- “isEnabledForNotification” on page 111
- “notifier” on page 115
- “notifyObservers” on page 116

Class Name	Header File	Implementation Variant
ISortedBag	isb.h	List
IGSortedBag	isb.h	List
ISortedBagAsList	isblst.h	List
IGSortedBagAsList	isblst.h	List
ISortedBagAsTable	isbtabs.h	Table
IGSortedBagAsTable	isbtabs.h	Table
ISortedBagAsDilTable	isbdil.h	Diluted table
IGSortedBagAsDilTable	isbdil.h	Diluted table

Sorted Bag

Members

All members of flat collections are described in Chapter 14, “Introduction to Flat Collections” on page 93. The following members are provided for sorted bag:

Method	Page	Method	Page
Constructor	97	isLast	111
Copy Constructor	97	lastElement	111
Destructor	97	locate	112
operator!=	97	locateNext	113
operator=	98	locateOrAdd	113
operator==	98	maxNumberOfElements	115
add	98	newCursor	115
addAllFrom	99	numberOfDifferentElements	116
addDifference	102	numberOfElements	116
addIntersection	103	numberOfOccurrences	116
addUnion	104	positionAt	117
allElementsDo	105	remove	117
anyElement	106	removeAll	118
compare	106	removeAllOccurrences	118
contains	106	removeAt	119
containsAllFrom	106	removeAtPosition	119
copy	107	removeFirst	120
differenceWith	108	removeLast	120
elementAt	108	replaceAt	120
elementAtPosition	109	setToFirst	121
firstElement	110	setToLast	121
intersectionWith	110	setToNext	122
isBounded	110	setToNextDifferentElement	122
isEmpty	110	setToPosition	123
isFirst	111	setToPrevious	123
isFull	111	unionWith	124

Sorted Bag also defines a cursor that inherits from `IOrderedCursor`. The members for `IOrderedCursor` are described in Chapter 37, “Cursor” on page 257.

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 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 *IBM Open Class Library User's Guide* for further information.

Sorted Bag

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

Chapter 31. 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 *IBM 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 *IBM 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 notifications with your collections, change the name of the desired collection class template in the list below from I... to IV....

Notification-enabled classes have the following additional members:

- “disableNotification” on page 108
- “enableNotification” on page 109
- “isEnabledForNotification” on page 111
- “notifier” on page 115
- “notifyObservers” on page 116

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	ism1st.h	List
IGSortedMapAsList	ism1st.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 Chapter 14, “Introduction to Flat Collections” on page 93. The following members are provided for sorted maps:

Method	Page	Method	Page
Constructor	97	isFull	111
Copy Constructor	97	isLast	111
Destructor	97	key	111
operator!=	97	lastElement	111
operator=	98	locate	112
operator==	98	locateElementWithKey	112
add	98	locateNext	113
addAllFrom	99	locateNextElementWithKey	113
addDifference	102	locateOrAdd	113
addIntersection	103	locateOrAddElementWithKey	114
addOrReplaceElementWithKey	103	maxNumberOfElements	115
addUnion	104	newCursor	115
allElementsDo	105	numberOfElements	116
anyElement	106	positionAt	117
compare	106	remove	117
contains	106	removeAll	118
containsAllFrom	106	removeAt	119
containsAllKeysFrom	107	removeAtPosition	119
containsElementWithKey	107	removeElementWithKey	119
copy	107	removeFirst	120
differenceWith	108	removeLast	120
elementAt	108	replaceAt	120
elementAtPosition	109	replaceElementWithKey	121
elementWithKey	109	setToFirst	121
firstElement	110	setToLast	121
intersectionWith	110	setToNext	122
isBounded	110	setToPosition	123
isEmpty	110	setToPrevious	123
isFirst	111	unionWith	124

Sorted map also defines a cursor that inherits from `IOrderedCursor`. The members for `IOrderedCursor` are described in Chapter 37, “Cursor” on page 257.

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

Sorted Map

- 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:

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 *IBM 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`, 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 dataset as the sample program.

See Appendix A, "Header Files for Collection Class Library Coding Examples" on page 587 for the code of the `dsur.h` file.

```
/*-----*\
dskusage.CPP - Disk Usage Information is handled using
               a Sorted Map and a Sorted Relation.
               *****
This program reads a file containing disk space usage
records 'DiskSpaceUR'. Each record is added into two
Equality Key Sorted Collections.

One of these collections, DSURbyName, is a Sorted Map.
The key is the name, which is unique.

The other collection, DSURbySpace, is a Sorted Relation.
The key is the space, which can occur multiple times.

Using the iteration method allElementsDo, both Collections
then print their contents in the sorted order.
\*/
```

Sorted Map

```
| Note: If we could be sure that there would never be two  
| identical records in the input data, it would be better  
| to use the corresponding collections that do not need  
| element equality. These are KeySorted Set and  
| KeySorted Bag instead of Sorted Map and  
| 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 <ism.h>
// Use the default Sorted Relation as is:
#include <isr.h>

int main (int argc, char* argv[])
{ char* fspec = "dsu.dat"; // Default for input file

  if (argc > 1)   fspec = argv[1];

  ifstream inputfile (fspec);
  if (!inputfile)
    errorExit(20, "Unable to open input file", fspec);

  ISortedMap <DiskSpaceUR, char*> dsurByName;
  ISortedMap <DiskSpaceUR, char*>::Cursor curByName (dsurByName);

  IGSortedRelation <DiskSpaceUR, int, DSURBySpaceOps>
    dsurBySpace;
  IGSortedRelation <DiskSpaceUR, int, DSURBySpaceOps>::Cursor
    curBySpace(dsurBySpace);

    // Read all records into dsurByName
  while (inputfile.good()) {
    DiskSpaceUR dsur (inputfile);
    if (dsur.isValid ()) {
      dsurByName.add (dsur);
      dsurBySpace.add (dsur);
    }
  }
  if (! inputfile.eof ())
    errorExit (39, "Error during read of", fspec);

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

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

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

  for (curBySpace.setToLast ();
    curBySpace.isValid ();
    curBySpace.setToPrevious ())
    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);
}
```

The program produces the following output:

All Disk Space Usage records sorted (ascending) by name:

./tmp	116
./tmp/.X11-unix	4
./tmp/inutmpK5kBM_	8
./tmp/inutmpKIoCKs	4
./tmp/inutmpMrQBQD	16
./tmp/inutmpQ4IBFe	16
./tmp/objrepos.inst	60
./usr/bin	38280
./usr/bin/c++	8988
./usr/bin/graf	864
./usr/ccs	7872
./usr/ccs/bin	924
./usr/ccs/lib	6944
./usr/etc	8
./usr/etc/yp	4
./usr/include	4532
./usr/include/DPS	116
./usr/include/IN	80
./usr/include/aixif	8
./usr/include/arpa	28
./usr/include/diag	44
./usr/include/em78	16
./usr/include/idl	176
./usr/include/isode	492
./usr/include/jfs	64
./usr/include/net	60
./usr/include/netinet	128
./usr/include/netiso	192
./usr/include/netns	52
./usr/include/nfs	36
./usr/include/protocols	24
./usr/include/rpc	80
./usr/include/rpcsvc	156
./usr/include/sys	1808
./usr/sbin	592
./usr/sbin/refer	52
./usr/sbin/spell	320
./usr/sbin/tty	168
./usr/lib	58132
./usr/lib/INed	492
./usr/lib/INnet	68
./usr/lib/acct	4
./usr/lib/asw	224
./usr/lib/boot	3520
./usr/lib/drivers	4316
./usr/lib/dwb	400
./usr/lib/dwm	600
./usr/lib/em78	736
./usr/lib/font	9124
./usr/lib/graf	16
./usr/lib/hcon	912
./usr/lib/inst_updt	2532
./usr/lib/instl	136
./usr/lib/learn	20
./usr/lib/libtermcap	72
./usr/lib/lpd	1264
./usr/lib/methods	1864
./usr/lib/mh	1224
./usr/lib/microcode	6632
./usr/lib/netsvc	124
./usr/lib/nls	8156
./usr/lib/objrepos	7448
./usr/lib/ps	2000
./usr/lib/ras	136
./usr/lib/sa	24
./usr/lib/security	96

Sorted Map

./usr/lib/struct	112
./usr/lib/uucp	4
./usr/lost+found	4
./usr/lpp	266820
./usr/lpp/DPS	4664
./usr/lpp/INed	36
./usr/lpp/SC	9232
./usr/lpp/X11	45840
./usr/lpp/X11dev	668
./usr/lpp/X11devEn_US	36
./usr/lpp/X11fnt	1260
./usr/lpp/X11mEn_US	72
./usr/lpp/X11rte	1648
./usr/lpp/aic	15128
./usr/lpp/aicmEn_US	52
./usr/lpp/bos	31220
./usr/lpp/bosadt	3420
./usr/lpp/bosext1	1552
./usr/lpp/bosext2	2256
./usr/lpp/bosinst	392
./usr/lpp/bosnet	2004
./usr/lpp/bosperf	716
./usr/lpp/bseiEn_US	28
./usr/lpp/bsl	144
./usr/lpp/bsmEn_US	212
./usr/lpp/bspiEn_US	28
./usr/lpp/bssiEn_US	36
./usr/lpp/cobolcmp	16
./usr/lpp/cobolrte	16
./usr/lpp/colormaps	24
./usr/lpp/cpp	48
./usr/lpp/diagnostics	4156
./usr/lpp/em78	80
./usr/lpp/em78mEn_US	52
./usr/lpp/fonts	44
./usr/lpp/gai	2084
./usr/lpp/gsl	560
./usr/lpp/hcon	252
./usr/lpp/hconmEn_US	52
./usr/lpp/hft	336
./usr/lpp/info	45568
./usr/lpp/integrator	96
./usr/lpp/jls	16
./usr/lpp/lpex	3632
./usr/lpp/lu0	420
./usr/lpp/mf	4664
./usr/lpp/ncs	452
./usr/lpp/nfs	136
./usr/lpp/pci	652
./usr/lpp/sna	1328
./usr/lpp/snamEn_US	52
./usr/lpp/snmpd	192
./usr/lpp/tcpip	464
./usr/lpp/tex	40964
./usr/lpp/txtfmt	776
./usr/lpp/vdi	2948
./usr/lpp/vdimEn_US	44
./usr/lpp/workbench	16240
./usr/lpp/workbenchmEn_US	76
./usr/lpp/x_st_mgr	32
./usr/lpp/x_st_mgrmEn_US	12
./usr/lpp/xd3	2564
./usr/lpp/xlC	10488
./usr/lpp/xlCbrs	52
./usr/lpp/xlCcmp	400
./usr/lpp/xlCmEn_US	52
./usr/lpp/xlCrte	40
./usr/lpp/xlCwkb	60
./usr/lpp/xlC	2920
./usr/lpp/xlccmp	76
./usr/lpp/xlfcmp	24
./usr/lpp/xlp	2520

./usr/lpp/xlpcmp	316
./usr/lpp/xlpcmpiEn_US	36
./usr/lpp/xlpcmpmEn_US	40
./usr/lpp/xlprte	40
./usr/lpp/xlprtemEn_US	40
./usr/lpp/xlprtemsg	20
./usr/sbin	1972
./usr/sbin	10196
./usr/sbin/acct	244
./usr/sbin/uucp	404
./usr/share	5908
./usr/share/dict	336
./usr/share/info	124
./usr/share/lib	3656
./usr/share/lpp	696
./usr/share/man	1092
./usr/ucb	152
./usr/usg	4

All Disk Space Usage records sorted (descending) by space:

./tmp/.X11-unix	4
./tmp/inutmpKIoCKs	4
./usr/lib/acct	4
./usr/lib/uucp	4
./usr/lost+found	4
./usr/usg	4
./usr/etc/yp	4
./tmp/inutmpK5kBM_	8
./usr/include/aixif	8
./usr/etc	8
./usr/lpp/x_st_mgrmEn_US	12
./tmp/inutmpQ4IBFe	16
./tmp/inutmpMrQBQD	16
./usr/lpp/cobolcmp	16
./usr/lpp/cobolrte	16
./usr/lpp/jls	16
./usr/include/em78	16
./usr/lib/graf	16
./usr/lpp/xlprtemsg	20
./usr/lib/learn	20
./usr/lpp/colormaps	24
./usr/lpp/xlfcmp	24
./usr/include/protocols	24
./usr/lib/sa	24
./usr/lpp/bspiEn_US	28
./usr/lpp/bseiEn_US	28
./usr/include/arpa	28
./usr/lpp/x_st_mgr	32
./usr/lpp/bssiEn_US	36
./usr/lpp/INed	36
./usr/lpp/X11deviEn_US	36
./usr/lpp/xlpcmpiEn_US	36
./usr/include/nfs	36
./usr/lpp/xlprte	40
./usr/lpp/xlprtemEn_US	40
./usr/lpp/xlpcmpmEn_US	40
./usr/lpp/xlCrte	40
./usr/lpp/fonts	44
./usr/lpp/vdimEn_US	44
./usr/include/diag	44
./usr/lpp/cpp	48
./usr/lpp/aicmEn_US	52
./usr/lpp/em78mEn_US	52
./usr/lpp/hconmEn_US	52
./usr/lpp/snamEn_US	52
./usr/lpp/xlCmEn_US	52
./usr/lpp/xlCbrs	52
./usr/include/netns	52
./usr/sbin/refer	52
./tmp/objrepos.inst	60

Sorted Map

./usr/lpp/xlCwkb	60
./usr/include/net	60
./usr/include/jfs	64
./usr/lib/INnet	68
./usr/lpp/X11mEn_US	72
./usr/lib/libtermcap	72
./usr/lpp/xlccmp	76
./usr/lpp/workbenchmEn_US	76
./usr/lpp/em78	80
./usr/include/IN	80
./usr/include/rpc	80
./usr/lpp/integrator	96
./usr/lib/security	96
./usr/lib/struct	112
./tmp	116
./usr/include/DPS	116
./usr/lib/netsvc	124
./usr/share/info	124
./usr/include/netinet	128
./usr/lpp/nfs	136
./usr/lib/instl	136
./usr/lib/ras	136
./usr/lpp/bsl	144
./usr/ucb	152
./usr/include/rpcsvc	156
./usr/sbin/tty	168
./usr/include/idl	176
./usr/lpp/snmpd	192
./usr/include/netiso	192
./usr/lpp/bsmEn_US	212
./usr/lib/asw	224
./usr/sbin/acct	244
./usr/lpp/hcon	252
./usr/lpp/xlpcmp	316
./usr/sbin/spell	320
./usr/lpp/hft	336
./usr/share/dict	336
./usr/lpp/bosinst	392
./usr/lpp/xlCcmp	400
./usr/lib/dwb	400
./usr/sbin/uucp	404
./usr/lpp/lu0	420
./usr/lpp/ncs	452
./usr/lpp/tcpip	464
./usr/include/isode	492
./usr/lib/INed	492
./usr/lpp/gsl	560
./usr/sbin	592
./usr/lib/dwm	600
./usr/lpp/pci	652
./usr/lpp/X11dev	668
./usr/share/lpp	696
./usr/lpp/bosperf	716
./usr/lib/em78	736
./usr/lpp/txtfmt	776
./usr/bin/graf	864
./usr/lib/hcon	912
./usr/ccs/bin	924
./usr/share/man	1092
./usr/lib/mh	1224
./usr/lpp/X11fnt	1260
./usr/lib/lpd	1264
./usr/lpp/sna	1328
./usr/lpp/bosext1	1552
./usr/lpp/X11rte	1648
./usr/include/sys	1808
./usr/lib/methods	1864
./usr/sbin	1972
./usr/lib/ps	2000
./usr/lpp/bosnet	2004
./usr/lpp/gai	2084
./usr/lpp/bosext2	2256

./usr/lpp/xlp	2520
./usr/lib/inst_updt	2532
./usr/lpp/xd3	2564
./usr/lpp/xlc	2920
./usr/lpp/vdi	2948
./usr/lpp/bosadt	3420
./usr/lib/boot	3520
./usr/lpp/lpex	3632
./usr/share/lib	3656
./usr/lpp/diagnostics	4156
./usr/lib/drivers	4316
./usr/include	4532
./usr/lpp/DPS	4664
./usr/lpp/mf	4664
./usr/share	5908
./usr/lib/microcode	6632
./usr/ccs/lib	6944
./usr/lib/objrepos	7448
./usr/ccs	7872
./usr/lib/nls	8156
./usr/bin/c++	8988
./usr/lib/font	9124
./usr/lpp/SC	9232
./usr/sbin	10196
./usr/lpp/xlC	10488
./usr/lpp/aic	15128
./usr/lpp/workbench	16240
./usr/lpp/bos	31220
./usr/bin	38280
./usr/lpp/tex	40964
./usr/lpp/info	45568
./usr/lpp/X11	45840
./usr/lib	58132
./usr/lpp	266820

Chapter 32. Sorted Relation

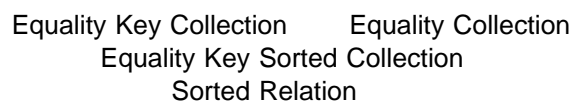
A *sorted relation* is an ordered collection of zero or more elements that have a key. The elements are sorted by the value of their key. Element equality is supported, and the values of the elements are relevant.

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 *IBM Open Class Library User's Guide* gives an overview of the properties of a sorted relation and its relationship to other flat collections.

Derivation



The diagram does not show all bases of sorted relation. See the figure “Abstract Class Hierarchy” in the *IBM 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 notifications with your collections, change the name of the desired collection class template in the list below from I... to IV....

Notification-enabled classes have the following additional members:

- “disableNotification” on page 108
- “enableNotification” on page 109
- “isEnabledForNotification” on page 111
- “notifier” on page 115
- “notifyObservers” on page 116

Sorted Relation

Class Name	Header File	Implementation Variant
ISortedRelation	isr.h	List
IGSortedRelation	isr.h	List
ISortedRelationAsList	isr1st.h	List
IGSortedRelationAsList	isr1st.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 Chapter 14, “Introduction to Flat Collections” on page 93. The following members are provided for sorted relation:

Method	Page	Method	Page
Constructor	97	key	111
Copy Constructor	97	lastElement	111
Destructor	97	locate	112
operator!=	97	locateElementWithKey	112
operator=	98	locateNext	113
operator==	98	locateNextElementWithKey	113
add	98	locateOrAdd	113
addAllFrom	99	locateOrAddElementWithKey	114
addDifference	102	maxNumberOfElements	115
addIntersection	103	newCursor	115
addOrReplaceElementWithKey	103	numberOfDifferentKeys	116
addUnion	104	numberOfElements	116
allElementsDo	105	numberOfElementsWithKey	116
anyElement	106	positionAt	117
compare	106	remove	117
contains	106	removeAll	118
containsAllFrom	106	removeAllElementsWithKey	118
containsAllKeysFrom	107	removeAt	119
containsElementWithKey	107	removeAtPosition	119
copy	107	removeElementWithKey	119
differenceWith	108	removeFirst	120
elementAt	108	removeLast	120
elementAtPosition	109	replaceAt	120
elementWithKey	109	replaceElementWithKey	121
firstElement	110	setToFirst	121
intersectionWith	110	setToLast	121
isBounded	110	setToNext	122
isEmpty	110	setToNextWithDifferentKey	122
isFirst	111	setToPosition	123
isFull	111	setToPrevious	123
isLast	111	unionWith	124

Sorted relation also defines a cursor that inherits from `IOrderedCursor`. The members for `IOrderedCursor` are described in Chapter 37, “Cursor” on page 257.

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

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>`

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 *IBM 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 211 for an example of a sorted relation.

Chapter 33. 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.

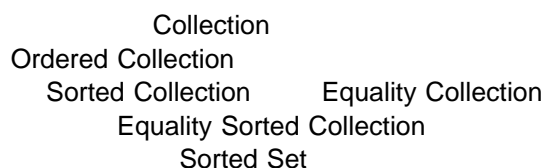
The elements of a sorted set are ordered such that the value of each element is less than or equal to the value of its successor.

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 *IBM 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 notifications with your collections, change the name of the desired collection class template in the list below from I... to IV....

Notification-enabled classes have the following additional members:

- “disableNotification” on page 108
- “enableNotification” on page 109
- “isEnabledForNotification” on page 111
- “notifier” on page 115
- “notifyObservers” on page 116

Class Name	Header File	Implementation Variant
ISortedSet	iss.h	AVL tree
IGSortedSet	iss.h	AVL tree

Sorted Set

Class Name	Header File	Implementation Variant
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 Chapter 14, “Introduction to Flat Collections” on page 93. The following members are provided for sorted sets:

Method	Page	Method	Page
Constructor	97	isFirst	111
Copy Constructor	97	isFull	111
Destructor	97	isLast	111
operator!=	97	lastElement	111
operator=	98	locate	112
operator==	98	locateNext	113
add	98	locateOrAdd	113
addAllFrom	99	maxNumberOfElements	115
addDifference	102	newCursor	115
addIntersection	103	positionAt	117
addUnion	104	remove	117
allElementsDo	105	removeAll	118
anyElement	106	removeAt	119
compare	106	removeAtPosition	119
contains	106	removeFirst	120
containsAllFrom	106	removeLast	120
copy	107	replaceAt	120
differenceWith	108	setToFirst	121
elementAt	108	setToLast	121
elementAtPosition	109	setToNext	122
firstElement	110	setToPosition	123
intersectionWith	110	setToPrevious	123
isBounded	110	unionWith	124
isEmpty	110		

Sorted Set also defines a cursor that inherits from `IOrderedCursor`. The members for `IOrderedCursor` are described in Chapter 37, “Cursor” on page 257.

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>
```

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

Sorted Set

- 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>
```

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

Element Type

- Copy constructor
- Destructor
- Assignment
- Ordering relation

Abstract Class

```
IASortedSet<Element>
```

For polymorphism, you can use the corresponding abstract class, `IASortedSet`, which is found in the `iass.h` header file. See the section on *Polymorphism and the Collections* in the *IBM 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 587 for the code of the `planet.h` file.

```
/*-----*\
| planets.CPP - All known planets are handled in a Sorted Set.          |
|           *****                                                    |
| This example creates several sorted sets of planets.                  |
| The sort order is based on each planets distance from                 |
| the sun.                                                                |
\*-----*/

#include <iostream.h>

// Let's use the Sorted Set Default Variant:
#include <iss.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 Neptun Pluto

Heavy Planets:

Jupiter Saturn Uranus Neptun

Bright Planets:

Mercury Venus Mars Jupiter

Bright-or-Heavy Planets:

Mercury Venus Mars Jupiter Saturn Uranus Neptun

Did you notice that all these Sets are sorted in the same order
(distance of planet from sun) ?

Chapter 34. 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 *IBM 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 notifications with your collections, change the name of the desired collection class template in the list below from I... to IV....

Notification-enabled classes have the following additional members:

- "disableNotification" on page 108
- "enableNotification" on page 109
- "isEnabledForNotification" on page 111
- "notifier" on page 115
- "notifyObservers" on page 116

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 Chapter 14, “Introduction to Flat Collections” on page 93. The following members are provided for stack:

Method	Page	Method	Page
Constructor	97	isFull	111
Copy Constructor	97	isLast	111
Destructor	97	lastElement	111
operator=	98	maxNumberOfElements	115
add	98	newCursor	115
addAllFrom	99	numberOfElements	116
addAsLast	100	pop	116
allElementsDo	105	positionAt	117
anyElement	106	push	117
compare	106	removeAll	118
copy	107	removeLast	120
elementAt	108	setToFirst	121
elementAtPosition	109	setToLast	121
firstElement	110	setToNext	122
isBounded	110	setToPosition	123
isEmpty	110	setToPrevious	123
isFirst	111	top	124

Stack also defines a cursor that inherits from `IOrderedCursor`. The members for `IOrderedCursor` are described in Chapter 37, “Cursor” on page 257.

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

- 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 *IBM 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.CPP - Simple example for the use of the Stack.          |
|                                                                |
\*-----*/

#include <string.h>
#include <iostream.h>
// Let's use the default stack: IStack
#include <istk.h>

typedef IStack <char*> SimpleStack;
// The stack requires iteration to be const.
typedef IConstantApplicator <char*> StackApplicator;

/*-----*\
* 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
{
public:
    IBoolean applyTo(char* const& w)
    {
        cout << w << endl;
        return(True);
    }
};

```

```

/*-----*\
* Main program                                     *
\*-----*/
int main()
{
    SimpleStack Stack1, Stack2;
    char *S;
    PrintClass Print;

    // We specify two stacks.
    // First all the strings are pushed onto the first stack.
    // Next, they are popped from the first and pushed onto
    // the second.
    // Finally they are popped from the second and printed.
    // During this final print the strings must appear
    // in their original order.

    int i;

    for (i = 0; i < 9; i++) {
        Stack1.push(String[i]);
    }

    cout << "Contents of Stack1:" << endl;
    Stack1.allElementsDo(Print);
    cout << "-----" << endl;

    while (!Stack1.isEmpty()) {
        Stack1.pop(S);           // Pop from stack 1
        Stack2.push(S);          // Add it on top of stack 2
    }

    cout << "Contents of Stack2:" << endl;
    Stack2.allElementsDo(Print);
    cout << "-----" << endl;

    while (!Stack2.isEmpty()) {
        Stack2.pop(S);
        cout << "Popped from Stack 2: " << S << endl;
    }

    return(0);
}

```

This program produces the following output:

```

Contents of Stack1:
The
quick
brown
fox
jumps
over
a
lazy
dog.
-----
Contents of Stack2:
dog.
lazy
a
over
jumps
fox
brown
quick
The
-----
Popped from Stack 2: The
Popped from Stack 2: quick

```

Stack

```
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

Chapter 35. Introduction to Trees	237
Defining the Traversal Order of Tree Elements	237
Chapter 36. Multiway Tree	239
Template Arguments and Required Functions	239
Terms Used	240
Coding Example for Multiway Tree	240
Tree Functions	244

Chapter 35. 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 Chapter 36, “Multiway Tree” on page 239.

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 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).

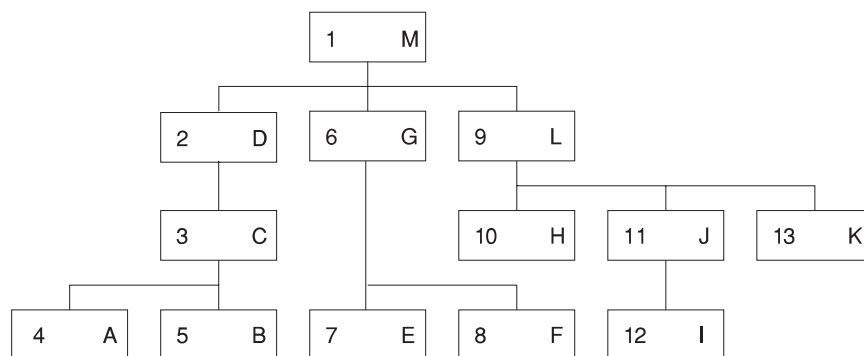


Figure 2. Preorder and Postorder Iteration Methods for Trees

Chapter 36. 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 an 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 descendents 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 "-".

```

/*****
*
*   Licensed Materials - Property of IBM
*
*   5645-001
*   (C) Copyright IBM Corp. 1992, 1997
*
*   US Government Users Restricted Rights - Use, duplication or
*   disclosure restricted by GSA ADP Schedule Contract with IBM
*   Corp.
*
*****/

#pragma csect (CODE, "EXPR")
#pragma csect (STATIC, "expr")

#pragma comment (copyright, \
"Licensed Materials - Property of IBM\n\n \
5645-001 \n \
(C) Copyright IBM Corp. 1992, 1997 \n\n \
US Government Users Restricted Rights - Use, duplication or\n \
disclosure restricted by GSA ADP Schedule Contract with IBM\n \
Corp.")

/*-----*\
| expr.CPP - An example of using a Multiway Tree
| Construct a tree for the following expression:
|           (8+2) * (2+4) / (7-5) ==> result: 30
| This is done explicitly for the following reasons:
| - no parser is available
| - program demonstrates the use of some common
|   functions for multiway trees.
| This program also calculates the result from the
| expression. A subtree (with two operands and one
| operator) is calculated and replaced by the result.
| Note that the code does not respect
| precedence of "/" and "*" over "+" and "-".
|-----*\

```

```
#include <imwt.h>
#include <istring.hpp>
#include <iostream.h>

//////////////////////////////////////
// The tree for this expression looks like follows: //
//                                                    //
//                                                    //
//                      /                            //
//                      //                          //
//                      *                            //
//                      -                            //
//                      //                          //
//                      +          +      7          5 //
//                      //                          //
//                      8    2      2    4            //
//////////////////////////////////////

typedef IMultiwayTree <2, IString> BinaryTree;

/***** functions *****/

IBoolean printNode(IString const& node, void* dummy)
/**** prints one node of a multiway tree ****/
{
    cout << node << "|";
    return True;
}

void prefixedNotation(BinaryTree const& binTree)
/* prints an binary tree in prefixed notation */
{
    binTree.allElementsDo(printNode , IPreorder);
    cout << endl;
}

void identifyChildren (IString &child1,
                     IString &child2,
                     BinaryTree &binTree,
                     ITreeCursor &binTreeCursor)
/***** identifies the children of a node *****/
{
    binTree.setToNext(binTreeCursor, IPreorder);
    child1 = binTree.elementAt(binTreeCursor);
    binTree.setToNextExistingChild(binTreeCursor);
    child2 = binTree.elementAt(binTreeCursor);
    binTree.setToParent(binTreeCursor);
}

IBoolean isNumber(IString child)
/**** checks whether a node contains a number *****/
{
    if ((child != '+' ) &&
        (child != '-' ) &&
        (child != '*' ) &&
        (child != '/'))
        { return True; }
    else { return False; }
}

void lookForNextOperator(BinaryTree &binTree,
                       ITreeCursor &binTreeCursor)
/**** looks for the next operator in the tree *****/
```

```

{
    IBoolean operatorFound = False;

    do
    {
        if (!isNumber(binTree.elementAt(binTreeCursor)))
        {
            operatorFound = True;
        }
        else
        {
            binTree.setToNext(binTreeCursor, IPreorder);
        }
    }
    while (! operatorFound);
}

void calculateSubtree(double &result, double &operand1,
                    double &operand2, IString &operatorSign)
/*****
**** calculates the result from a subtree in the ****
**** complete tree ****
*****/
{
    switch (*(char*)operatorSign)
    {
        case '+':
            result = operand1+operand2;
            break;
        case '-':
            result = operand1-operand2;
            break;
        case '/':
            result = operand1/operand2;
            break;
        case '*':
            result = operand1*operand2;
            break;
    } /* endswitch */
}

/***** main *****/
int main ()
{
    //////////////////////////////////////
    // Constructing 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 like follows:
            //save the cursor, because it will become invalidated after
            //removeSubtree
            binTreeSaveCursor = binTreeCursor;
            binTree.setToParent(binTreeSaveCursor);
            replacePosition = binTree.position(binTreeCursor);
            binTree.removeSubtree(binTreeCursor);
            binTree.addAsChild(binTreeSaveCursor,
                replacePosition,
                (IString)result);
            cout << "Tree with calculated subtree replaced: "
                << endl;
            prefixedNotation(binTree);
            binTree.setToRoot(binTreeCursor);
        }
        else
        {
            // if tree contains root with two children only,replace
            // this calculated subtree by its result like 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);

```

Tree Collection Functions

```
    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`

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
- IChildAlreadyExistsException

addAsRoot

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

Adds the given element as root of the tree.

Precondition: The tree must not have a root; that is, it must be empty.

Exceptions

- IOutOfMemory
- IRootAlreadyExistsException

allElementsDo, allSubtreeElementsDo

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

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

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

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

Calls the given function for all elements of the subtree denoted by the given cursor (of this tree) until the given function returns false. The elements are visited in the

given iteration order. Additional arguments can be passed to the given function using *additionalArgument*. The additional argument defaults to zero if no additional argument is given. The `allElementsDo()` function (without a subtree cursor argument) iterates over all elements of the tree.

Note: The given function must not remove elements from or add elements to the tree.

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

Preconditions

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

Exception: `ICursorInvalidException`

allElementsDo, allSubtreeElementsDo

```
IBoolean allElementsDo (
    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`

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`

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.

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.
- $(1 \leq \textit{position} \leq \textit{numberOfChildren})$

Exceptions

- ICursorInvalidException
- IPositionInvalidException

isEmpty

```
IBoolean isEmpty ( ) const;
```

Returns true if the tree is empty.

isLeaf

```
IBoolean isLeaf ( ITreeCursor const& cursor ) const;
```

Returns true if the node pointed to by the given cursor is a leaf node of the tree. A leaf node is a node with no children.

Precondition: The cursor must point to an element of this tree.

Exception: ICursorInvalidException

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

Tree Collection Functions

numberOfChildren

INumber **numberOfChildren** () const;

Returns the number of children a node can possibly have. The actual number of children of any node will always be less than or equal to this number.

numberOfElements, numberOfSubtreeElements

INumber **numberOfElements** () const;

INumber **numberOfSubtreeElements** (
ITreeCursor const& *cursor*) const;

Returns the number of elements that the subtree denoted by the given cursor contains. The subtree root, inner, and leaf nodes are counted. The `numberOfElements()` function (without a cursor argument) counts the number of elements in the whole tree.

Preconditions: The cursor must belong to the tree and must point to an element in the tree.

Exception: `ICursorInvalidException`

numberOfLeaves, numberOfSubtreeLeaves

INumber **numberOfLeaves** () const;

INumber **numberOfSubtreeLeaves** (
ITreeCursor const& *cursor*) const;

Returns the number of leaf elements that the subtree denoted by the given cursor contains. Leaves are nodes that have no children. The `numberOfLeaves()` function (without a cursor argument) counts the number of leaves in the whole tree.

Preconditions: The cursor must belong to the tree and must point to an element in the tree.

Exception: `ICursorInvalidException`

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

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

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

setToNextExistingChild

```
IBoolean setToNextExistingChild (  
    ITreeCursor& cursor ) const;
```

Sets the cursor to the next existing sibling of the node denoted by the given cursor (of this tree). Invalidates the cursor if the node has no next sibling. A node with no next sibling is the last existing child of its parent.

Precondition: The cursor must point to an element of this tree.

Return Value: Returns true if the node has a next sibling.

Exception: ICursorInvalidException

setToParent

`IBoolean setToParent (ITreeCursor& cursor) const;`

Sets the cursor to the parent of the node denoted by the given cursor (of this tree). Invalidates the cursor if the node has no parent. A node with no parent is the root node of its tree.

Precondition: The cursor must point to an element of this tree.

Return Value: Returns true if the node has a parent.

Exception: `ICursorInvalidException`

setToPrevious

`IBoolean setToPrevious (ITreeCursor& cursor,
ITreeIterationOrder iterationOrder) const;`

Sets the cursor to the previous node in the given iteration order. Invalidates the cursor if there is no previous node.

Precondition: The cursor must point to an element of this tree.

Return Value: Returns true if the given cursor does not point to the first node (in iteration order).

Exception: `ICursorInvalidException`

setToPreviousExistingChild

`IBoolean setToPreviousExistingChild (ITreeCursor& cursor) const;`

Sets the cursor to the previous existing sibling of the node denoted by the given cursor (of this tree). Invalidates the cursor if the node has no previous sibling. A node with no previous sibling is the first existing child of its parent.

Precondition: The cursor must point to an element of this tree.

Return Value: Returns true if the node has a previous sibling.

Exception: `ICursorInvalidException`

setToRoot

`IBoolean setToRoot (ITreeCursor& cursor) const;`

Sets the cursor to the root node of the tree. Invalidates the cursor if the tree is empty (that is, if no root node exists).

Precondition: The cursor must belong to this tree.

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

Exception: `ICursorInvalidException`

Part 5. Auxiliary Collection Classes

This part describes the auxiliary collection classes. The auxiliary classes are those classes which support other classes, and which include classes for cursors, pointers, and iterators.

The auxiliary classes are discussed in the following chapters:

Chapter 37. Cursor	257
Public Member Functions	258
Chapter 38. Tree Cursor	261
Public Members of Tree Cursor	261
Chapter 39. Applicator and Constant Applicator Classes	265
Chapter 40. Pointer Classes	267
Members	267
Coding Example for Managed Element Pointer	269
Chapter 41. Collection Event Data	273
Members	273
Chapter 42. Collection Guard	275
Members	275
Chapter 43. Restricted Access Collection Guard	277
Members	277
Chapter 44. Tree Collection Guard	279
Members	279

Chapter 37. 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	258	operator==	258
copy	258	setToFirst	259
isValid	258	setToLast	259
invalidate	258	setToNext	259
element	258	setToPrevious	259
operator!=	258		

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.

operator==

IBoolean **operator==** (ICursor const& cursor) const;

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

setToFirst

`IBoolean setToFirst () ;`

Sets the cursor to the first element of the associated collection in iteration order. Invalidates the cursor if the collection is empty (if no first element exists).

Return Value: Returns true if the associated collection is not empty.

setToLast

`IBoolean setToLast () ;`

Sets the cursor to the last element of the associated collection in iteration order. Invalidates the cursor if the collection is empty (no last element exists). This function is only available for cursors of ordered collections. Returns true if the associated collection was not empty.

setToNext

`IBoolean setToNext () ;`

Sets the cursor to the next element in the associated collection in iteration order. Invalidates the cursor if no more elements are left to be visited. Returns true if there was a next element.

Precondition: The cursor must point to an element of the associated collection.

Exception: `ICursorInvalidException`

setToPrevious

`IBoolean setToPrevious () ;`

Sets the cursor to the previous element of the associated collection in iteration order. Invalidates the cursor if no such element exists. This function is only available for cursors of ordered collections.

Return Value: Returns true if a previous element exists.

Precondition: The cursor must point to an element of the associated collection.

Exception: `ICursorInvalidException`

Chapter 38. 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>	262
<code>operator==</code>	261	<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>	263
<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.

`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.

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 true 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`

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.

Chapter 39. 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 105 for more information on this function.) Iteration stops when `applyTo()` returns `false`.

The figure *Iteration Using `allElementsDo`* in the *IBM Open Class Library User's Guide* explains the concepts and usage of iterations.

Derivation

These classes do not derive from any other class.

Header File

`iiter.h`

Members

These classes define only one function, as a virtual function.

`applyTo`

```
virtual IBoolean applyTo (Element const& element) = 0;
```

This function applies a series of specified statements or a function to all elements of a collection for which you use the 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 *IBM Open Class Library User's Guide*.

Chapter 40. 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 *IBM 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	267	Conversion operator	268
Copy constructor	268	operator->	268
Destructors	268	operator=	268
operator*	268	operator==	268

Constructors

```
IAutoPointer ();
IElemPointer ();
IMngPointer ();
```

Constructs a pointer of the indicated type and initializes it with `NULL`.

Constructors from a Given C++ Pointer

```
IAutoPointer (Element *ptr, IExplicitInit)
IAutoElemPointer (Element *ptr, IExplicitInit)
IElemPointer (Element *ptr, IExplicitInit = IINIT)
IMngPointer (Element *ptr, IExplicitInit)
IMngElemPointer (Element *ptr, IExplicitInit)
```

Constructs a pointer object of the indicated type from a given C++ pointer. For managed pointers, the reference count of the referenced element is set to 1.

Copy Constructors from a Given Collection Class Pointer

```
IAutoPointer (IAutoPointer < Element > const& ptr)
```

```
IMngPointer (IMngPointer < Element > const& ptr)
```

Constructs a new pointer and initializes it with the given pointer. For automatic pointers, the given pointer is set to NULL. For managed pointers, the reference count of the referenced element is incremented by 1.

Destructors

```
~IAutoPointer ()
```

```
~IAutoElemPointer ()
```

Deletes the object referenced to by the automatic pointer.

```
~IMngPointer ()
```

```
~IMngElemPointer ()
```

Destructs the pointer and decrements the reference count of the referenced element. If the reference count is 0, the referenced element is deleted.

operator*

```
Element& operator * () const;
```

Returns a reference to the object to which the pointer refers.

Conversion operator

```
operator Element* () const
```

Implicitly convert this pointer to a C++ pointer.

operator->

```
Element* operator-> () const
```

Returns a C pointer to the object to which the pointer refers.

operator=

```
void operator = (IAutoPointer < Element > const& ptr)  
IMngPointer < Element >& operator = (IMngPointer < Element > const& ptr)  
IMngElemPointer < Element >& operator = (IMngElemPointer < Element > const& ptr)
```

Assigns the given pointer to this pointer. For automatic pointers, the given pointer is set to NULL and the previously referenced element is deleted. For managed pointers, the reference count of the referenced element is incremented and the reference count of the previously referenced element is decremented.

operator==

The pointer classes do not have an `operator==` explicitly defined for them. However, for equality test you can use the syntax:

```
pointerVariable1 == pointerVariable2;
```

The conversion operator (`operator Element*`) implicitly converts the objects to C pointers, and then the `operator==` for C pointers is invoked.

Because the operator== is not actually a member of the class, you cannot write an equality test like the following:

```
if (pointerVariable1.operator==(pointerVariable2)) { /* ... */ }
```

Coding Example for Managed Element Pointer

The following sample allows you to store managed pointers for various graphical objects into a key sorted set. The graphical objects, namely lines, curves, and circles, inherit from a base class Graphics. Using these pointers, you can draw the various shapes from the collection.

```

/*****
 *
 * Licensed Materials - Property of IBM
 *
 * 5645-001
 * (C) Copyright IBM Corp. 1992, 1997
 *
 * US Government Users Restricted Rights - Use, duplication or
 * disclosure restricted by GSA ADP Schedule Contract with IBM
 * Corp.
 *
 *****/

#pragma csect (CODE, "GRAPH")
#pragma csect (STATIC, "graph")

#pragma comment (copyright, \
"Licensed Materials - Property of IBM\n\n \
5645-001 \n \
(C) Copyright IBM Corp. 1992, 1997 \n\n \
US Government Users Restricted Rights - Use, duplication or\n \
disclosure restricted by GSA ADP Schedule Contract with IBM\n \
Corp.")

/*-----*\
| graph.CPP - demonstrate how to use collection class pointers |
|               *****                                       |
| Different graphical shapes (curves, circles, lines)          |
| inherit from an abstract class "graph".                     |
| Managed element pointers for these different shapes are      |
| stored in one collection, a key sorted set.                 |
| These pointers are used to draw the different shapes.       |
\*-----*/

#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);

    forICursor(gpsCursor)
    {

```

```

        drawObject = gpsCursor.element();

        sout << endl
              << " Key is: " << drawObject->graphicsKey()
              << endl
              << " ID is: " << drawObject->id()
              << endl;

        drawObject->draw();

    } /* endfor */

    return sout;
}

int main ()
{
    MngPointerKSet graphMngPointerKSet;

    /******
    /* Adding 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: "
    << endl
    << graphMngPointerKSet
    << endl << " " << endl;

graphMngPointerKSet.elementAtWithKey(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;

}

```

The program produces the following output:

Drawing the shapes from the key set of Managed Pointers:

```

Key is: 10
ID is: Curve 1
drawing Curve 1
with starting point: (1.1|4.3)
and with fix points: (2.1|6.4)(3.1|9.7)(4.1|6.5)
and with ending point: (5.1|7.4)

Key is: 20
ID is: Curve 2
drawing Curve 2
with starting point: (1.2|3.9)
and with fix points: (2.2|5.9)(3.2|8.8)(4.2|7.5)
and with ending point: (5.2|9.4)

Key is: 40
ID is: Circle 1
drawing Circle 1
with center: (1|1) and with radius: 1

Key is: 50
ID is: Circle 2
drawing Circle 2
with center: (2|2) and with radius: 2

Key is: 70
ID is: Line 1
drawing Line 1
with starting point: (1.1|1.1) and with ending point: (5.1|5.1)

Key is: 80
ID is: Line 2
drawing Line 2
with starting point: (2.2|2.2) and with ending point: (5.2|5.2)

Key is: 90
ID is: Line 3
drawing Line 3
with starting point: (3.3|3.3) and with ending point: (5.3|5.3)

drawing Line 1
with starting point: (1.1|1.1) and with ending point: (5.1|5.1)
drawing Line 3
with starting point: (3.3|3.3) and with ending point: (5.3|5.3)
drawing Line 3

```

Pointer Classes

```
with starting point: (3.3|3.3) and with ending point: (5.3|5.3)
Curve 1 will now be deleted ...
Circle 1 will now be deleted ...
Curve 2 will now be deleted ...
Line 1 will now be deleted ...
Line 3 will now be deleted ...
Line 2 will now be deleted ...
Circle 2 will now be deleted ...
```

Chapter 41. Collection Event Data

Derivation

Inherits from none.

Inherited By

None.

Header File

iiycllct.h

Class Name

IVCollectionEventData

Members

Member	Page
cursor	273
element	273

This class provides support for notifications. The collection notifications addId, removeId, and replaceId pass a pointer to this class.

Members

cursor

```
ICursor const&  
  cursor() const;
```

Use this function to access the collection element being modified, when the notification addId, removeId, or replaceId is sent. This function returns a cursor that points to the modified element (the added element, the removed element, or the replacing element).

element

```
Element const * const  
  element() const;
```

Use this function to access a collection element that has been replaced, when the notification replaceId is sent. This function returns a cursor that points to the replaced element.

Chapter 42. Collection Guard

Derivation

Inherits from none.

Inherited By

None.

Header File

iacclt.h

Class Name

ICollectionGuard

Members

Member	Page
Constructor	275
Destructor	275
Use objects of the ICollectionGuard class to ensure the thread safety of a collection in a multithreaded program.	
The guard object makes sure that only one thread at a time accesses a collection, by locking the collection and preventing other threads from accessing it.	
The guard object must be destroyed prior to the destruction of the corresponding collection.	

Members

Constructor

```
ICollectionGuard( IACollection < Element >&,
                 long timeout = - 1 );
```

The constructor takes the collection object to be locked and an optional timeout value as parameters. The timeout value is specified in milliseconds. If a lock request cannot be resolved within the specified range of time, an exception is thrown. The timeout value defaults to -1. On POSIX systems this parameter is ignored.

Destructor

```
~ICollectionGuard();
```

Unlocks the collection specified within the constructor of this guard.

Chapter 43. Restricted Access Collection Guard

Derivation

Inherits from none.

Inherited By

None.

Header File

iarstrct.h

Class Name

IRestrictedAccessCollectionGuard

Members

Member	Page
Constructor	277
Destructor	277

Use objects of the IRestrictedAccessCollectionGuard class to ensure the thread safety of a collection in a multithreaded program.

The guard object makes sure that only one thread at a time accesses a collection, by locking the collection and preventing other threads from accessing it.

The guard object must be destroyed prior to the destruction of the corresponding collection.

Members

Constructor

```
IRestrictedAccessCollectionGuard(
    IRestrictedAccessCollection < Element >&,
    long timeout = - 1 );
```

The constructor takes the collection object to be locked and an optional timeout value as parameters. The timeout value is specified in milliseconds. If a lock request cannot be resolved within the specified range of time, an exception is thrown. The timeout value defaults to -1. On POSIX systems this parameter is ignored.

Destructor

```
~IRestrictedAccessCollectionGuard();
```

Unlocks the collection specified within the constructor of this guard.

Chapter 44. Tree Collection Guard

Derivation

Inherits from none.

Inherited By

None.

Header File

iatree.h

Class Name

ITreeCollectionGuard

Members

Member	Page
Constructor	279
Destructor	279

Use objects of the ITreeCollectionGuard class to ensure the thread safety of a tree collection in a multithreaded program.

The guard object makes sure that only one thread at a time accesses a collection, by locking the collection and preventing other threads from accessing it.

The guard object must be destroyed prior to the destruction of the corresponding collection.

Members

Constructor

```
ITreeCollectionGuard(
    IATree < Element >&,
    long timeout = - 1 );
```

The constructor takes the collection object to be locked and an optional timeout value as parameters. The timeout value is specified in milliseconds. If a lock request cannot be resolved within the specified range of time, an exception is thrown. The timeout value defaults to -1. On POSIX systems this parameter is ignored.

Destructor

```
~ITreeCollectionGuard();
```

Unlocks the collection specified within the constructor of this guard.

Part 6. Abstract 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.

The abstract classes are discussed in the following chapters:

Chapter 45. Collection	283
Chapter 46. Equality Collection	285
Chapter 47. Equality Key Collection	287
Chapter 48. Equality Key Sorted Collection	289
Chapter 49. Equality Sorted Collection	291
Chapter 50. Key Collection	293
Chapter 51. Key Sorted Collection	295
Chapter 52. Ordered Collection	297
Chapter 53. Sequential Collection	299
Chapter 54. Sorted Collection	301
Chapter 55. Restricted Access Collection	303

Chapter 45. 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 *IBM 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 Chapter 14, “Introduction to Flat Collections” on page 93. The following member functions are provided for collection:

Method	Page	Method	Page
Destructor	97	isFull	111
add	98	maxNumberOfElements	115
addAllFrom	99	newCursor	115
anyElement	106	numberOfElements	116
copy	106	removeAll	118
elementAt	97	removeAt	119
elementAtPosition	109	replaceAt	120
isBounded	110	setToFirst	121
isEmpty	110	setToNext	122

Chapter 46. 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 *IBM 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 `iequal.h`.

Members

The equality collection class defines the following member functions, described in Chapter 14, “Introduction to Flat Collections” on page 93, as virtual functions:

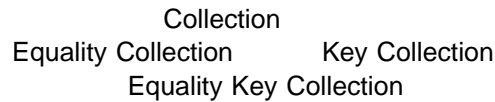
Method	Page	Method	Page
Destructor	97	locateOrAdd	113
contains	106	numberOfOccurrences	116
containsAllFrom	106	remove	117
locate	112	removeAllOccurrences	118
locateNext	113		

Chapter 47. 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 *IBM 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.

Chapter 48. 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 *IBM 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 *IBM 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

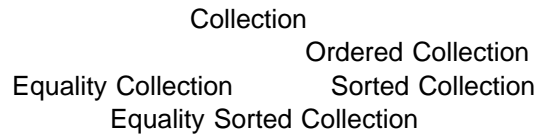
All the members of equality key sorted collection are inherited from its base classes.

Chapter 49. 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 *IBM 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.

Chapter 50. 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 *IBM 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 Chapter 14, “Introduction to Flat Collections” on page 93, as virtual functions:

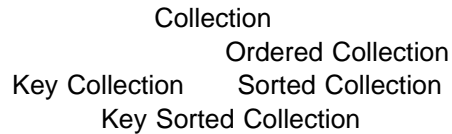
Method	Page	Method	Page
Destructor	97	locateOrAddElementWithKey	114
addOrReplaceElementWithKey	103	numberOfDifferentKeys	116
containsAllKeysFrom	107	numberOfElementsWithKey	116
containsElementWithKey	107	removeAllElementsWithKey	118
elementWithKey	109	removeElementWithKey	119
key	111	replaceElementWithKey	121
locateElementWithKey	112	setToNextWithDifferentKey	122
locateNextElementWithKey	113		

Chapter 51. 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 *IBM 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.

Chapter 52. 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 *IBM 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 Chapter 14, “Introduction to Flat Collections” on page 93, as pure virtual functions:

Method	Page	Method	Page
Destructor	97	removeAtPosition	119
elementAtPosition	109	removeFirst	120
firstElement	110	removeLast	120
isFirst	111	setToLast	121
isLast	111	setToPosition	123
lastElement	111	setToPrevious	123
position	117		

Chapter 53. 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 *IBM 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	97	isFull	111
operator=	98	isLast	111
add	98	lastElement	111
addAllFrom	99	maxNumberOfElements	115
addAsFirst	100	newCursor	115
addAsLast	100	position	117
addAsNext	101	removeAll	118
addAsPrevious	101	removeAt	119
addAtPosition	101	removeAtPosition	119
allElementsDo	105	removeFirst	120
anyElement	106	removeLast	120
compare	106	replaceAt	120
elementAt	108	setToFirst	121
elementAtPosition	109	setToLast	121
firstElement	110	setToNext	122
isBounded	110	setToPosition	123
isEmpty	110	setToPrevious	123
isFirst	111	sort	123

Chapter 54. 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 *IBM 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.

Chapter 55. Restricted Access Collection

Because Restricted Access Collection is an abstract class, it cannot be used to create any objects. Restricted Access Collection defines the interfaces for the restricted access collections stack, queue, and deque.

Derivation

ICollectionBase
IARestrictedAccessCollection

Inherited By

IADeque
IAQueue
IAStack

Header File

iarstrct.h

Class Name

IARestrictedAccessCollection

Members

Member	Page
Constructor	97
Destructor	97
add	98
addAllFrom	99
allElementsDo	105
any	106
copy	107
disableNotification	108
elementAt	108
elementAtPosition	109
enableNotification	109
first	110
isBounded	110
isEmpty	110
isEnabledForNotification	111
isFirstAt	111
isFull	111
isLastAt	111
last	111
maxNumberOfElements	115
newCursor	115
notifier	115
notifyObservers	116
numberOfElements	116
positionAt	117
removeAll	118
setToFirst	121
setToLast	121
setToNext	122
setToPosition	123
setToPrevious	123

Part 7. Application Support Class Library

Chapter 56. Base Classes	311
IBase	311
IVBase	313
Chapter 57. Buffer Classes	315
IBuffer	315
Nested Type Definitions	325
IDBCSBuffer	325
Chapter 58. IDate Class	335
Chapter 59. Exception Classes	341
IException	341
Public Data	347
Nested Classes	347
Nested Type Definitions	348
Protected Members	348
IAccessError	349
IAssertionFailure	350
ICLibErrorInfo	351
IDeviceError	353
IBaseErrorInfo	354
IException::TraceFn	357
IExceptionLocation	358
IGUIErrorInfo	359
IInvalidParameter	362
IInvalidRequest	363
IMessageText	364
IOutOfMemory	366
IOutOfSystemResource	367
IOutOfWindowResource	368
IResourceExhausted	369
ISystemErrorInfo	371
IXLibErrorInfo	373
Chapter 60. String Classes	377
IString	377
IOString	405
IStringEnum	412
IStringParser	413
IStringParser::SkipWords	416
IStringTest	417
IStringTestMemberFn	418
Chapter 61. IApplication	421
Public Functions	421
Inherited Public Functions	424
Protected Functions	424
Inherited Protected Data	425

Chapter 62. Decimal Classes	427
IBinaryCodedDecimal	427
Public Members	430
Protected Functions	435
Decimal	436
Public Members	438
Chapter 63. ICurrentApplication	443
Public Functions	443
Inherited Public Functions	445
Protected Functions	445
Inherited Protected Functions	446
Inherited Protected Data	446
Chapter 64. ICurrentThread	447
Public Functions	448
Inherited Public Functions	454
Protected Functions	455
Inherited Protected Functions	455
Inherited Protected Data	456
Chapter 65. IEnumHandle	457
Public Functions	457
Nested Type Definitions	458
Chapter 66. IEventData	459
Public Functions	460
Inherited Public Functions	463
Inherited Protected Data	463
Chapter 67. IEventParameter1	465
Inherited Public Functions	465
Inherited Protected Data	465
Chapter 68. IEventParameter2	467
Inherited Public Functions	467
Inherited Protected Data	467
Chapter 69. IEventResult	469
Inherited Public Functions	469
Inherited Protected Data	469
Chapter 70. IHandle	471
Public Functions	471
Inherited Public Functions	472
Protected Data	472
Inherited Protected Data	473
Nested Type Definitions	473
Chapter 71. IHighEventParameter	475
Inherited Public Functions	475
Inherited Protected Data	475
Chapter 72. ILowEventParameter	477

Inherited Public Functions	477
Inherited Protected Data	477
Chapter 73. INotificationEvent	479
Public Functions	479
Inherited Public Functions	481
Inherited Protected Data	482
Chapter 74. INotifier	483
Public Functions	484
Inherited Public Functions	485
Protected Functions	485
Inherited Protected Data	486
Chapter 75. IObserver	487
Public Functions	487
Inherited Public Functions	488
Protected Functions	488
Inherited Protected Data	489
Chapter 76. IObserverList	491
Public Functions	491
Inherited Public Functions	493
Inherited Protected Data	493
Nested Classes	494
Chapter 77. IObserverList::Cursor	495
Public Functions	495
Inherited Public Functions	497
Inherited Protected Data	497
Chapter 78. IPrivateResource	499
Public Functions	500
Inherited Public Functions	500
Inherited Protected Functions	501
Inherited Protected Data	501
Chapter 79. IPrivateSemaphoreHandle	503
Public Functions	503
Nested Type Definitions	504
Chapter 80. IProcessId	505
Public Functions	505
Nested Type Definitions	506
Chapter 81. IRefCounted	507
Public Functions	507
Inherited Public Functions	508
Protected Functions	508
Inherited Protected Data	509
Chapter 82. IReference	511
Public Functions	512
Inherited Public Functions	513

Inherited Protected Data	513
Chapter 83. IResource	515
Public Functions	515
Inherited Public Functions	516
Inherited Protected Data	517
Chapter 84. IResourceLock	519
Public Functions	519
Inherited Public Functions	520
Protected Functions	520
Inherited Protected Data	521
Chapter 85. ISharedResource	523
Public Functions	524
Inherited Public Functions	525
Inherited Protected Functions	525
Inherited Protected Data	525
Chapter 86. ISharedSemaphoreHandle	527
Public Functions	527
Nested Type Definitions	528
Chapter 87. IStandardNotifier	529
Public Functions	529
Inherited Public Functions	531
Protected Functions	531
Inherited Protected Functions	533
Public Data	533
Inherited Protected Data	533
Chapter 88. IThread	535
Public Functions	537
Inherited Public Functions	551
Protected Functions	552
Inherited Protected Data	552
Nested Classes	552
Nested Type Definitions	553
Chapter 89. IThread::Cursor	555
Public Functions	555
Inherited Public Functions	557
Inherited Protected Data	557
Chapter 90. IThreadFn	559
Public Functions	559
Inherited Public Functions	560
Inherited Protected Data	560
Chapter 91. IThreadHandle	561
Public Functions	561
Public Data	562
Nested Type Definitions	562

Chapter 92. IThreadId	563
Public Functions	563
Chapter 93. IThreadMemberFn	565
Public Functions	565
Inherited Public Functions	566
Inherited Protected Data	567
Chapter 94. ITime Class	569
Chapter 95. ITimeStamp	573
Public Functions	573
Public Data	577
Chapter 96. ITrace Class	579

Chapter 56. Base Classes

IBase

Derivation

IBase

Inherited By

IDate
IEventData
IHandle
IReference
IString
ITime
ITimeStamp
IVBase

Header File

ibase.hpp

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 for IBase in other 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 Application Support Class Library declares in isynonym.hpp.

Nested Classes

IBase::Version

Public Members

asDebugInfo

IString **asDebugInfo()** const;

Obtains the diagnostic version of an object's contents.

asString

IString **asString()** const;

Obtains the standard version of an object's contents.

messageFile

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

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

OS/2-Specific Information: If you previously called `setMessageFile` (see page 313) 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 (`dde4uile.msg`).

OS/390 C++-Specific Information: Messages specific to the Application Support Class Library and Collection Class Library are retrieved using Language Environment Message Handling Services. Because message catalogs are not supported in the MVS environment, you cannot create your own messages and merge them with the Class Library messages.

messageText

```
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);
```

Returns the message text associated with the specified message ID. You can specify up to nine optional text strings to insert into the message.

OS/2-Specific Information: If the message is found in a message segment that has been bound to the `.exe`, the message is loaded from the application. Otherwise, the message is searched for in the message file described before. The search order for this file is as follows:

- The system root directory
- The current working directory
- Using the `DPATH` environment setting
- Using the `APPEND` environment setting

operator<<

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

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 you should override this function in all derived classes.

setMessageFile

```
static void setMessageFile(const char *msgFileName);
```

Sets the message file from which the class library loads its exception text. The name must include the file extension.

OS/390 C++-Specific Information: The exception text for the standard set of exceptions defined in the Application Support Class Library and Collection Class Library is loaded using Language Environment Message Handling Services. The file name set through this function is not used on MVS. The function is supported for portability only.

version

```
static Version version();
```

Returns the Application Support 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.

Enumerations

BooleanConstants

```
public:
enum BooleanConstants { false = 0, true = 1 };
```

The Application Support 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.

IVBase

Derivation

```
IBase
IVBase
```

Header File

```
ivbase.hpp
```

The IVBase class provides basic generic behavior for all Application Support Class Library classes that have virtual functions. In addition, this class enables derived classes to exploit the nested type and value names in the class IBase (see page 311), such as Boolean, true, and false.

Your derived classes might need to override the virtual functions IVBase::asString (see page 314) and IVBase::asDebugInfo (see page 314). This enables automatic support for the output of derived class objects to ostreams, such as cout, cerr, or both.

Public Members

asDebugInfo

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

Obtains the diagnostic version of an object's contents.

asString

```
virtual IString asString() const;
```

Obtains the standard version of an object's contents.

operator<<

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

Allows any library object to be dumped to an ostream, such as:

```
cout << anObject;
```

This function is a left-shift operator.

Inherited Public Members

Member	Class	Page	Member	Class	Page
asDebugInfo	IBase	311	operator<<	IBase	312
asString	IBase	311			

Chapter 57. Buffer Classes

IBuffer

Derivation

```

IBase
  IVBase
    IBuffer
  
```

Header File

```
ibuffer.hpp
```

Objects of the IBuffer class define the contents of an IString (see page 377).

Constructors

```

protected:
IBuffer(unsigned newLen);
  
```

The constructor for this class requires the length of the buffer, which is the value to be stored in the len data member.

Public Members

addRef

```
void addRef();
```

Increments the usage count.

asDebugInfo

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

Returns information about the buffer's internal representation that you can use for debugging.

center

```
virtual IBuffer *center(unsigned newLen, char padCharacter);
```

Centers the receiver within a string of the specified length.

change

```

virtual IBuffer *change(const char *pSource, unsigned sourceLen,
                        const char *pTarget, unsigned targetLen,
                        unsigned startPos, unsigned numChanges);
  
```

Changes occurrences of a specified pattern to a specified replacement string. You can specify the number of changes to perform. The default is to change all occurrences of the pattern. You can also specify the position in the receiver at which to begin.

The parameters are the following:

pSource The pattern string as null-terminated string. The library searches for the pattern string within the receiver's data.

sourceLen
The length of the source string.

pTarget The target string as a null-terminated string. It replaces the occurrences of the pattern string in the receiver's data.

targetLen The length of the target string.

startPos The position to start the search at within the target's data.

numChanges
The number of patterns to search for and change.

charType

```
virtual IStringEnum::CharType charType(unsigned index) const;
```

Returns the type of a character at the specified index.

checkAddition

```
static unsigned checkAddition(unsigned addend1, unsigned addend2);
```

Verifies that the two parameters, when added, do not overflow an unsigned integer.

checkMultiplication

```
static unsigned checkMultiplication(unsigned factor1, unsigned factor2);
```

Verifies that the two parameters, when multiplied, do not overflow an unsigned integer.

compare

```
virtual Comparison compare(const void *p, unsigned len) const;
```

Compares the buffer's contents to the contents of the specified character array.

contents

```
const char *contents() const;  
char *contents();
```

Returns the address of the buffer's contents.

copy

```
virtual IBuffer *copy(unsigned numCopies);
```

Replaces the receiver's contents with a specified number of replications of itself.

defaultBuffer

```
static IBuffer *defaultBuffer();
```

Returns the address of the null buffer for the class. This is a static function.

fromContents

```
static IBuffer *fromContents(const char *pBuffer);
```

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 Application Support Class Library can only return values from the contents function of this class. Otherwise, if the returned IBuffer pointer is used, errors could occur.

includesDBCS

```
virtual Boolean includesDBCS() const;
```

If any characters are DBCS (double-byte character set), true is returned.

includesMBCS

```
virtual Boolean includesMBCS() const;
```

If any characters are MBCS (multiple-byte character set), true is returned.

includesSBCS

```
virtual Boolean includesSBCS() const;
```

If any characters are SBCS (single-byte character set), true is returned.

indexOf

```
virtual unsigned indexOf(const char *pString, unsigned len, unsigned startPos = 1) const;
```

```
virtual unsigned indexOf(const IStringTest &aTest, unsigned startPos = 1) const;
```

Returns the byte index of the first occurrence of the specified string within the receiver. If there are no occurrences, 0 is returned.

indexOfAnyBut

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

```
virtual unsigned indexOfAnyBut(const char *pString, unsigned len,  
                               unsigned startPos = 1) const;
```

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 (see page 417) object.

indexOfAnyOf

```
virtual unsigned indexOfAnyOf(const char *pString, unsigned len,  
                              unsigned startPos = 1) const;
```

```
virtual unsigned indexOfAnyOf(const IStringTest &aTest, unsigned startPos = 1) const;
```

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 (see page 417) object.

IBuffer

insert

```
virtual IBuffer *insert(const char *pInsert, unsigned insertLen,  
                        unsigned pos, char padCharacter);
```

Inserts the specified string after the specified location.

isAlphabetic

```
virtual Boolean isAlphabetic() const;
```

If all the characters are in {'A'-'Z','a'-'z'}, true is returned. A value of false is returned for a null input string.

isAlphanumeric

```
virtual Boolean isAlphanumeric() const;
```

If all the characters are in {'A'-'Z','a'-'z','0'-'9'}, true is returned. A value of false is returned for a null input string.

isASCII

```
virtual Boolean isASCII() const;
```

If all the characters are in {0x00-0x7F}, true is returned. A value of false is returned for a null input string.

isControl

```
virtual Boolean isControl() const;
```

Returns true if all the characters are control characters. A value of false is returned for a null input string.

Control characters are determined using the iscntrl() 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}.

isDBCS

```
virtual Boolean isDBCS() const;
```

If all the characters are DBCS, true is returned. A value of false is returned for a null input string.

isDigits

```
virtual Boolean isDigits() const;
```

If all the characters are in {'0'-'9'}, true is returned. A value of false is returned for a null input string.

isGraphics

```
virtual Boolean isGraphics() const;
```

Returns true if all the characters are graphics characters. A value of false is returned for a null input string.

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}.

isHexDigits

```
virtual Boolean isHexDigits() const;
```

If all the characters are in {'0'-'9','A'-'F','a'-'f'}, true is returned. A value of false is returned for a null input string.

isLowerCase

```
virtual Boolean isLowerCase() const;
```

If all the characters are in {'a'-'z'}, true is returned. A value of false is returned for a null input string.

isMBCS

```
virtual Boolean isMBCS() const;
```

If all the characters are MBCS, true is returned. A value of false is returned for a null input string.

isPrintable

```
virtual Boolean isPrintable() const;
```

Returns true if all the characters are printable characters. A value of false is returned for a null input string.

Printable characters are defined by the isprint() 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}.

isPunctuation

```
virtual Boolean isPunctuation() const;
```

If none of the characters is white space, a control character, or an alphanumeric character, true is returned. A value of false is returned for a null input string.

isSBCS

```
virtual Boolean isSBCS() const;
```

If all the characters are SBCS, true is returned. A value of false is returned for a null input string.

isUpperCase

```
virtual Boolean isUpperCase() const;
```

If all the characters are in {'A'-'Z'}, true is returned. A value of false is returned for a null input string.

isValidDBCS

```
virtual Boolean isValidDBCS() const;
```

If no DBCS characters have a 0 second byte, true is returned. A value of false is returned for a null input string.

isValidMBCS

```
virtual Boolean isValidMBCS() const;
```

If no MBCS characters have a 0 second byte, true is returned. A value of false is returned for a null input string.

isWhiteSpace

```
virtual Boolean isWhiteSpace() const;
```

Returns true if all the characters are whitespace characters. A value of false is returned for a null input string.

Whitespace characters are defined by the isspace() function as defined in the space locale source file and in the space class of the LC_CTYPE category of the current locale. For example, on ASCII systems, printable characters are those in the range {0x09-0x0D,0x20}.

lastIndexOf

```
virtual unsigned lastIndexOf(const IStringTest &aTest, unsigned startPos = 0) const;  
virtual unsigned lastIndexOf(const char *pString, unsigned len,  
                             unsigned startPos = 0) const;
```

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 $0 \leq x \leq \text{startPos}$. 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.

lastIndexOfAnyBut

```
virtual unsigned lastIndexOfAnyBut(const char *pString, unsigned len,  
                                  unsigned startPos = 0) const;  
virtual unsigned lastIndexOfAnyBut(const IStringTest &aTest,  
                                  unsigned startPos = 0) const;
```

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.

lastIndexOfAnyOf

```
virtual unsigned lastIndexOfAnyOf(
    const IStringTest &aTest, unsigned startPos = 0) const;

virtual unsigned lastIndexOfAnyOf(const char *pString, unsigned len,
    unsigned startPos = 0) const;
```

Returns the index of the last character in the specified string or character. The search starts at the position specified by *startPos* (inclusive) and proceeds backward. The default of 0 starts the search at the end of the receiver's string. If the search target is not found, 0 is returned.

If you specify 0 for *startPos*, this function returns 0, indicating the search target was not found.

leftJustify

```
virtual IBuffer *leftJustify(unsigned newLen, char padCharacter);
```

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.

length

```
unsigned length() const;
```

Returns the length of the buffer's contents.

lowerCase

```
virtual IBuffer *lowerCase();
```

Translates all uppercase letters in the receiver to lowercase.

newBuffer

```
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;
```

Allocates a new buffer and initializes it with the contents of up to three specified buffers.

The parameters are the following:

- | | |
|-------------|---|
| <i>p1</i> | The pointer to the first part to be copied into the data area of the new buffer. The first part is <i>len1</i> bytes long. If the pointer is null, the <i>padChar</i> is copied for <i>len1</i> bytes. |
| <i>len1</i> | The length, in bytes, of the first part to be copied into the new buffer. |
| <i>p2</i> | 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 <i>len2</i> bytes long. If the pointer is null, the <i>padChar</i> is copied for <i>len2</i> bytes. If nothing is specified for <i>p2</i> , it is null. |
| <i>len2</i> | The length, in bytes, of the second part to be copied into the new buffer. If nothing is specified for <i>len2</i> , it defaults to 0 bytes. |

IBuffer

- 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.

next

```
virtual const char *next(const char *prev) const;  
virtual char *next(const char *prev);
```

Returns a pointer to the next character, not the next byte, in the buffer.

null

```
IBuffer *null() const;
```

Returns the address of the null buffer.

overflow

```
static unsigned overflow();
```

Throws an exception when `IBuffer::checkAddition` (see page 316) or `IBuffer::checkMultiplication` (see page 316) detect an overflow.

overlayWith

```
virtual IBuffer *overlayWith(const char *overlay, unsigned len,  
                             unsigned pos, char padCharacter);
```

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*).

remove

```
virtual IBuffer *remove(unsigned startPos, unsigned numChars);
```

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.

removeRef

```
void removeRef();
```

Decrements the usage count and deletes the buffer when the usage count goes to 0.

reverse

```
virtual IBuffer *reverse();
```

Reverses the receiver's contents.

rightJustify

```
virtual IBuffer *rightJustify(unsigned newLen, char padCharacter);
```

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.

setDefaultBuffer

```
static void setDefaultBuffer(IBuffer *newDefaultBuffer);
```

Sets the default (null) buffer. The specified buffer must be comprised of a single null byte.

strip

```
virtual IBuffer *strip(const IStringTest &aTest, IStringEnum::StripMode mode);
virtual IBuffer *strip(const char *pChars, unsigned len, IStringEnum::StripMode mode);
```

Strips both leading and trailing character or characters. You can specify the character or characters as the following:

- A `char*` array
- An `IStringTest` (see page 417) object

The default is white space.

subString

```
virtual IBuffer *subString(unsigned startPos, unsigned len, char padCharacter) const;
```

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 object for this class is returned.
- padCharacter* Specifies the character the function uses to pad the copied string if fewer than *len* bytes have been copied from the source buffer.

IBuffer

translate

```
virtual IBuffer *translate(const char *pInputChars, unsigned inputLen,  
                           const char *pOutputChars, unsigned outputLen,  
                           char padCharacter);
```

Converts all of the receiver's characters that are in the first specified string to the corresponding character in the second specified string.

upperCase

```
virtual IBuffer *upperCase();
```

Translates all lowercase letters in the receiver to uppercase.

useCount

```
unsigned useCount() const;
```

Returns the number of IStrings referring to the buffer.

Inherited Public Members

Member	Class	Page	Member	Class	Page
asDebugInfo	IBase	311	asString	IVBase	314
asDebugInfo	IVBase	314	operator<<	IBase	312
asString	IBase	311	operator<<	IVBase	314

Protected Members

allocate

```
virtual IBuffer *allocate(unsigned bufLength) const;
```

Returns a new buffer of the specified length.

className

```
virtual const char *className() const;
```

Returns the name of the class (IBuffer).

initialize

```
static IBuffer *initialize();
```

Initializes (sets up a null buffer, a DBCS table, and so forth). This is a static function.

operator delete

```
void operator delete(void *p);
```

Deallocates a buffer.

operator new

```
void *operator new(size_t t, unsigned bufLen);
```

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*.

startBackwardsSearch

```
virtual unsigned startBackwardsSearch(unsigned startPos, unsigned searchLen) const;
```

Initializes a search of type `IString::lastIndexOf` (see page 388).

- 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.

startSearch

```
virtual unsigned startSearch(unsigned startPos, unsigned searchLen) const;
```

Initializes a search of type `IString::indexOf` (see page 383).

- 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.

Nested Type Definitions
Comparison

```
typedef enum { equal , greaterThan , lessThan } Comparison;
```

These enumerators specify the possible valid return codes from `IBuffer::compare` (p. 316).

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.

IDBCSBuffer
Derivation

```
IBase
IVBase
IBuffer
IDBCSBuffer
```

Header File

idbcsbuf.hpp

Objects of the IDBCSBuffer class implement the version of IString (see page 377) 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.

Constructors

```
protected:  
IDBCSBuffer(unsigned bufLength);
```

The constructor for this class is protected. Only IDBCSBuffer::allocate (see page 326) and IBuffer::initialize (see page 324) can call the constructor.

Public Members

allocate

```
IBuffer *allocate(unsigned newLen) const;
```

Returns a new buffer of the specified length.

center

```
IBuffer *center(unsigned newLen, char padCharacter);
```

Centers the receiver within a string of the specified length.

charType

```
IStringEnum::CharType charType(unsigned index) const;
```

Returns the type of a character at the specified index.

includesDBCS

```
Boolean includesDBCS() const;
```

If any characters are DBCS (double-byte character set), true is returned.

includesMBCS

```
Boolean includesMBCS() const;
```

If any characters are MBCS (multiple-byte character set), true is returned.

includesSBCS

```
Boolean includesSBCS() const;
```

If any characters are SBCS (single-byte character set), true is returned.

indexOf

```
unsigned indexOf(const char *pString, unsigned len, unsigned startPos = 1) const;
unsigned indexOf(const IStringTest &aTest, unsigned startPos = 1) const;
```

Returns the byte index of the first occurrence of the specified string within the receiver. If there are no occurrences, 0 is returned.

indexOfAnyBut

```
unsigned indexOfAnyBut(const IStringTest &aTest, unsigned startPos = 1) const;
unsigned indexOfAnyBut(const char *pString, unsigned len, unsigned startPos = 1) const;
```

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 (see page 417) object.

indexOfAnyOf

```
unsigned indexOfAnyOf(const IStringTest &aTest, unsigned startPos = 1) const;
unsigned indexOfAnyOf(const char *pString, unsigned len, unsigned startPos = 1) const;
```

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 (see page 417) object.

insert

```
IBuffer *insert(const char *pInsert, unsigned insertLen,
               unsigned pos, char padCharacter);
```

Inserts the specified string after the specified location.

isDBCS

```
Boolean isDBCS() const;
```

If all the characters are DBCS, true is returned. A value of false is returned for a null input string.

isMBCS

```
Boolean isMBCS() const;
```

If all the characters are MBCS, true is returned. A value of false is returned for a null input string.

isSBCS

```
Boolean isSBCS() const;
```

If all the characters are SBCS, true is returned. A value of false is returned for a null input string.

isValidDBCS

```
Boolean isValidDBCS() const;
```

If no DBCS characters have a 0 second byte, true is returned. A value of false is returned for a null input string.

isValidMBCS

```
Boolean isValidMBCS() const;
```

If no MBCS characters have a 0 second byte, true is returned. A value of false is returned for a null input string.

lastIndexOf

```
unsigned lastIndexOf(const char *pString, unsigned len, unsigned startPos = 0) const;  
unsigned lastIndexOf(const IStringTest &aTest, unsigned startPos = 1) const;
```

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 $0 \leq x \leq \text{startPos}$. 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.

lastIndexOfAnyBut

```
unsigned lastIndexOfAnyBut(const IStringTest &aTest, unsigned startPos = 0) const;  
unsigned lastIndexOfAnyBut(const char *pString, unsigned len, unsigned startPos = 0) const;
```

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.

lastIndexOfAnyOf

```
unsigned lastIndexOfAnyOf(const char *pString, unsigned len, unsigned startPos = 0) const;  
unsigned lastIndexOfAnyOf(const IStringTest &aTest, unsigned startPos = 0) const;
```

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.

leftJustify

```
IBuffer *leftJustify(unsigned newLen, char padCharacter);
```

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.

lowerCase

```
IBuffer *lowerCase();
```

Translates all uppercase letters in the receiver to lowercase.

next

```
char *next(const char *prev);
const char *next(const char *prev) const;
```

Returns a pointer to the next character, not the next byte, in the buffer.

overlayWith

```
IBuffer *overlayWith(const char *overlay, unsigned len,
                    unsigned pos, char padCharacter);
```

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*).

remove

```
IBuffer *remove(unsigned startPos, unsigned numChars);
```

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.

reverse

```
IBuffer *reverse();
```

Reverses the receiver's contents.

rightJustify

```
IBuffer *rightJustify(unsigned newLen, char padCharacter);
```

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.

strip

```
IBuffer *strip(const char *pChars, unsigned len, IStringEnum::StripMode mode);
IBuffer *strip(const IStringTest &aTest, IStringEnum::StripMode mode);
```

Strips both leading and trailing character or characters. You can specify the character or characters as the following:

- A char* array
- An IStringTest (see page 417) object

The default is white space.

subString

```
IBuffer *subString(unsigned startPos, unsigned len, char padCharacter) const;
```

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 object for this class 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.

translate

```
IBuffer *translate(const char *pInputChars, unsigned inputLen,  
                  const char *pOutputChars, unsigned outputLen,  
                  char padCharacter);
```

Converts all of the receiver's characters that are in the first specified string to the corresponding character in the second specified string.

upperCase

```
IBuffer *upperCase();
```

Translates all lowercase letters in the receiver to uppercase.

Inherited Public Members

Member	Class	Page	Member	Class	Page
addRef	IBuffer	315	isControl	IBuffer	318
asDebugInfo	IBase	311	isDBCS	IBuffer	318
asDebugInfo	IVBase	314	isDigits	IBuffer	318
asDebugInfo	IBuffer	315	isGraphics	IBuffer	318
asString	IBase	311	isHexDigits	IBuffer	319
asString	IVBase	314	isLowerCase	IBuffer	319
center	IBuffer	315	isMBCS	IBuffer	319
change	IBuffer	315			
charType	IBuffer	316			
compare	IBuffer	316			
contents	IBuffer	316			
copy	IBuffer	316			
includesDBCS	IBuffer	317			
includesMBCS	IBuffer	317			
includesSBCS	IBuffer	317			
indexOf	IBuffer	317			
indexOfAnyBut	IBuffer	317			
indexOfAnyOf	IBuffer	317			
insert	IBuffer	318			
isAlphabetic	IBuffer	318			
isAlphanumeric	IBuffer	318			
isASCII	IBuffer	318			

Member	Class	Page
isPrintable	IBuffer	319
isPunctuation	IBuffer	319
isSBCS	IBuffer	319
isUpperCase	IBuffer	319
isValidDBCS	IBuffer	320
isValidMBCS	IBuffer	320
isWhiteSpace	IBuffer	320
lastIndexOf	IBuffer	320
lastIndexOfAnyBut	IBuffer	320
lastIndexOfAnyOf	IBuffer	321
leftJustify	IBuffer	321
length	IBuffer	321
lowerCase	IBuffer	321
newBuffer	IBuffer	321
next	IBuffer	322
null	IBuffer	322
operator<<	IBase	312
operator<<	IVBase	314
overlayWith	IBuffer	322
remove	IBuffer	322
removeRef	IBuffer	322
reverse	IBuffer	323
rightJustify	IBuffer	323
strip	IBuffer	323
subString	IBuffer	323
translate	IBuffer	324
upperCase	IBuffer	324
useCount	IBuffer	324

Protected Members

charLength

```
static size_t charLength(char const* pChar);
size_t charLength(unsigned pos) const;
size_t charLength(unsigned pos, mbstate_t* pMBState ) const;
static size_t charLength( char const* pChar, mbstate_t* pMBState );
```

Returns the number of bytes in the character whose first byte is pointed to by char *. This is a static function.

className

```
const char *className() const;
```

Returns the name of the class (IDBCSBuffer).

isCharValid

```
Boolean isCharValid(unsigned pos, const char *pValidChars, unsigned numValidChars) const;
```

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.

isDBCS1

```
Boolean isDBCS1(unsigned pos) const;
```

If the byte at the specified offset is the first byte of DBCS, true is returned.

Note:

This member is available on OS/2 only.

The Application Support Class Library provides this function only for compatibility with prior library versions. We recommend using `IDBCSBuffer::charLength` (see page 331) to determine if the byte is part of a multiple-byte character.

isPrevDBCS

```
Boolean isPrevDBCS(unsigned pos) const;
```

If the preceding character to the one at the specified offset is a DBCS character, true is returned.

Note:

This member is available on OS/2 only.

The Application Support Class Library provides this function only for compatibility with prior library versions. We recommend using `IDBCSBuffer::prevCharLength` (see page 332) to determine if the preceding byte is part of a multiple-byte character.

isSBC

```
static Boolean isSBC( char const* pChar );  
static Boolean isSBC( char const* pChar, mbstate_t* pMBState );
```

If the byte pointed to by the specified character is a single-byte character, true is returned. This is a static function.

maxCharLength

```
static size_t maxCharLength();
```

Returns the maximum number of bytes in a multiple-byte character. This is a static function.

prevCharLength

```
size_t prevCharLength(unsigned pos) const;  
size_t prevCharLength(unsigned pos, mbstate_t* pMBState) const;
```

Returns the number of bytes in the preceding character to the one at the specified offset.

startBackwardsSearch

```
unsigned startBackwardsSearch(unsigned startPos, unsigned searchLen) const;
```

Initializes a search of type `IString::lastIndexOf` (see page 388).

- 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.

startSearch

```
unsigned startSearch(unsigned startPos, unsigned searchLen) const;
```

Initializes a search of type `IString::indexOf` (see page 383).

- 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.

Inherited Protected Members

Member	Class	Page	Member	Class	Page
allocate	IBuffer	324	operator new	IBuffer	324
className	IBuffer	324	startBackwardsSearch	IBuffer	325
operator delete	IBuffer	324	startSearch	IBuffer	325

Chapter 58. lDate Class

Derivation

```
lBase
    lDate
```

Header File

```
ldate.hpp
```

Objects of the lDate class represent specified dates. This class also provides general day and date-handling functions. Externally, dates consist of three pieces of information:

- A year
- A month within that year
- A day within that month

The Application Support Class Library also lets you specify the day within the year.

The lDate class returns locale-sensitive information, such as names of days and months, in the current locale defined at runtime. See the description of the standard C function `setlocale` in the *OS/390 C/C++ Run-Time Library Reference* and *OS/390 C/C++ Programming Guide* for information about setting the locale.

Constructors

```
public:
lDate();
lDate(Month aMonth, int aDay, int aYear);
lDate(int aDay, Month aMonth, int aYear);
lDate(int aYear, int aDay);
lDate(const lDate &aDate);
lDate(unsigned long julianDayNumber);
```

Note: The following version of this constructor is available on OS/2 only.

```
public:
lDate(const _CDATE &cDate);
```

You can construct objects of this class by:

- Using the default constructor, which returns the current day.
- Giving the year, month, and day for the desired day. These parameters can be in either month/day/year or day/month/year order.
- Giving the year and day of the year for the desired day.
- Using `lDate::today` (see page 339) to return the current date.
- Copying another lDate object.
- Giving the Julian day number, as a long.
- Giving a container details CDATE structure.

Public Members

asCDATE

```
_CDate asCDate() const;
```

Returns a container CDATE structure for the date.

Note: This member is available on OS/2 only.

asString

```
String asString(const char *fmt) const;
String asString(YearFormat yearFmt = yy) const;
```

Returns the IDate as a string. The default is formatted per the system (mm-dd-yy). The other 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 list. ITime::asString (see page 570) provides the conversion specifiers that apply to ITime. For more information about the strftime function, refer to the *OS/390 C/C++ Run-Time Library Reference*.

Conversion Specifiers

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")
```

dayName

```
static NSString *dayName(NSInteger aDay);
NSString *dayName() const;
```

Returns the name of the receiver's day of the week:

- The first version of `dayName` accepts a specified day. It returns the name of the day of the week that is equivalent to the index value in *aDay*.
- The second version of `dayName` accepts no parameters. It returns the name of the receiver's day of the week, such as "Monday."

dayOfMonth

```
int dayOfMonth() const;
```

Returns the day in the receiver's month as an integer from 1 to 31.

dayOfWeek

```
NSInteger dayOfWeek() const;
```

Returns the index of the receiver's day of the week: Monday through Sunday.

dayOfYear

```
int dayOfYear() const;
```

Returns the day in the receiver's year as an integer from 1 to 366.

daysInMonth

```
static int daysInMonth(NSInteger aMonth, NSInteger aYear);
```

Returns the number of days in a specified month of a specified year. You must specify *aYear* in yyyy format.

daysInYear

```
static int daysInYear(NSInteger aYear);
```

Returns the number of days in a specified year. You must specify *aYear* in yyyy format.

isLeapYear

```
static Boolean isLeapYear(NSInteger aYear);
```

If the specified year is a leap year, true is returned. Otherwise, false is returned. You must specify *aYear* in yyyy format.

isValid

```
static Boolean isValid(NSInteger aMonth, NSInteger aDay, NSInteger aYear);
static Boolean isValid(NSInteger aDay, NSInteger aMonth, NSInteger aYear);
static Boolean isValid(NSInteger aYear, NSInteger aDay);
```

Indicates whether the specified date is valid. You must specify *aYear* in yyyy format. You can specify the date as:

- month/day/year
- day/month/year
- year/day

For example, February 29, 1990 is not a valid date because February only had 28 days in 1990.

IDate

julianDate

```
unsigned long julianDate() const;
```

Returns the Julian day number of the receiver IDate. This function uses the true definition of a Julian date; that is, it returns the number of days from January 1, 4713 B.C.

monthName

```
static IString monthName(Month aMonth);  
IString monthName() const;
```

Returns the name of the receiver's month:

- The first version of this function accepts no parameters. It returns the name of the receiver's month, such as "March."
- The second version of this function accepts a specified month. It returns the name of the month that is equivalent to the index value in *aMonth*.

monthOfYear

```
Month monthOfYear() const;
```

Returns the index of the receiver's month of the year: January through December.

operator!=

```
Boolean operator!=(const IDate &aDate) const;
```

If the IDate objects represent different dates, true is returned.

operator+

```
IDate operator+(int numDays) const;
```

Adds an integral number of days to the left-hand operand, yielding a new IDate.

operator+=

```
IDate &operator+=(int numDays);
```

Adds an integral number of days to the left-hand operand, assigning the result to that operand.

operator-

```
long operator-(const IDate &aDate) const;  
IDate operator-(int numDays) const;
```

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.

operator-=

```
IDate &operator-=(int numDays);
```

Subtracts an integral number of days from the right-hand operand, assigning the result to that operand.

operator<

```
Boolean operator<(const IDate &aDate) const;
```

If the left-hand operand represents a date prior to the date represented by the right-hand operand, true is returned.

operator<<

```
friend ostream &operator<<(ostream &aStream, const IDate &aDate);
```

Puts an IDate representation to an output stream. It puts the default IDate::asString representation.

operator<=

```
Boolean operator<=(const IDate &aDate) const;
```

If the left-hand operand represents a date prior to or identical to the date represented by the right-hand operand, true is returned.

operator==

```
Boolean operator==(const IDate &aDate) const;
```

If the IDate objects represent the same date, true is returned.

operator>

```
Boolean operator>(const IDate &aDate) const;
```

If the left-hand operand represents a date subsequent to the date represented by the right-hand operand, true is returned.

operator>=

```
Boolean operator>=(const IDate &aDate) const;
```

If the left-hand operand represents a date subsequent to or identical to the date represented by the right-hand operand, true is returned.

today

```
static IDate today();
```

Returns the current date. This is a static function.

year

```
int year() const;
```

Returns the receiver's year. The returned value is in the yyyy format.

Inherited Public Members

Member	Class	Page	Member	Class	Page
asDebugInfo	IBase	311	operator<<	IBase	312
asString	IBase	311			

Protected Members

initialize

```
IDate &initialize(Month aMonth, int aDay, int aYear);
```

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.

Enumerations

DayOfWeek

```
public:
typedef enum { Monday = 0, Tuesday, Wednesday, Thursday, Friday,
              Saturday, Sunday } DayOfWeek;
```

A typedef providing the values Monday through Sunday for the days of the week.

Month

```
public:
typedef enum { January = 1, February, March, April,
              May, June, July, August,
              September, October, November, December } Month;
```

A typedef providing the values January through December for the months of the year.

YearFormat

```
public:
typedef enum { yy, yyyy } YearFormat;
```

A typedef specifying the number of digits in the year for the default asString format (yy or yyyy).

Chapter 59. Exception Classes

IException

Derivation

IException

Header File

iexcbase.hpp

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 Application Support 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:

Class	Page
IAccessError	349
IAssertionFailure	350
IDeviceError	353
IInvalidParameter	362
IInvalidRequest	363
IResourceExhausted	369

In addition, IResourceExhausted has the following derived classes:

Class	Page
IOutOfMemory	366
IOutOfSystemResource	367
IOutOfWindowResource	368

Note: OS/390 C/C++ does not support IOutOfWindowResource. It is listed here because versions of the Application Support Class Library on other operating systems do support it. You can also derive your own exception type from IException.

The Application Support 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 re-throwing the exception.

IASSERT

If you define IC_DEVELOP during the compile for debugging purposes, this macro expands to provide assertion support in the library. This macro accepts an expression to test. If the test evaluation returns false, IASSERT calls assertParameter (see page 344).

IXCLASSDECLARE

Creates a declaration for a derived class of IException or one of its derived classes.

IXCLASSIMPLEMENT

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 (see page 358).

INO_EXCEPTIONS_SUPPORT

Supports compilers lacking an exception-handling implementation. If you use the INO_EXCEPTIONS_SUPPORT macro, the following macros end the program after capturing the location information and logging it. These macros normally throw an exception.

ITHROW	Found in IException.
IASSERTPARM	Found in IBaseErrorInfo (see page 354).
IASSERTSTATE	Found in IBaseErrorInfo.
ITHROWERROR	Found in IBaseErrorInfo.
ITHROWERROR1	Found in IBaseErrorInfo.
ITHROWLIBRARYERROR	Found in IBaseErrorInfo.
ITHROWLIBRARYERROR1	Found in IBaseErrorInfo.
ITHROWGUIERROR	Found in IGUIErrorInfo (see page 359).
ITHROWGUIERROR2	Found in IGUIErrorInfo.
ITROWSYSTEMERROR	Found in ISystemErrorInfo (see page 371).

Note: OS/390 C/C++ does not support the IGUIErrorInfo class. It is listed here because versions of the Application Support Class Library on other operating systems do support it. The INO_EXCEPTIONS_SUPPORT macro might not work correctly on all compilers.

Whenever the Application Support Class Library throws one of these exceptions, trace records are output with information about the exception. The class ITrace (see page 579) describes tracing in more detail.

Nested Classes

IException::TraceFn

Constructors

```
public:
IException(const char *errorText, unsigned long errorId = 0,
             Severity severity = IException : : unrecoverable);
IException(const IException &exception);
```

You can construct objects of this class by doing the following:

- Using the primary constructor. Normally, this is the only way you can construct an object of this class.

errorText The text describing this error.

errorId (Optional) The identifier you want to associate with this particular error.

severity (Optional) Use the enumeration IException::Severity (see page 346) to specify the severity of the error. The default is unrecoverable.

- Using the copy constructor. The Application Support Class Library provides this constructor so the compiler can copy the exception when it is thrown.

exception The exception object you want to copy.

Error Code

Use these members to determine which class library an exception originated from.

errorCodeGroup

Returns the error group the exception originated from.

```
ErrorCodeGroup
errorCodeGroup() const;
```

setErrorCodeGroup

Sets the ID of the originating class library into the exception object.

```
IException&
setErrorCodeGroup( ErrorCodeGroup errorGroup );
```

Public Members

addLocation

```
virtual IException &addLocation(const IExceptionLocation &location);
```

Adds the location information to the exception object. The Application Support Class Library captures this information when an exception is thrown or re-thrown. An array of IExceptionLocation objects is stored in the exception object.

Exception

location An IExceptionLocation (see page 358) object containing the following:

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

appendText

```
IException &appendText(const char *errorText);
```

Appends the specified text to the text string on the top of the exception text stack.

errorText The text you want to append.

assertParameter

```
static void assertParameter(const char *exceptionText,  
                           IExceptionLocation location);
```

The IASSERT macro uses this function to do the following:

1. Create an IAssertionFailure (see page 350) 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 (see page 358) object containing the following:

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

errorId

```
unsigned long errorId() const;
```

Returns the error ID of the exception.

isRecoverable

```
virtual int isRecoverable() const;
```

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.

locationAtIndex

```
const IExceptionLocation *locationAtIndex(unsigned long locationIndex) const;
```

Returns the IExceptionLocation (see page 358) object at the specified index.

locationIndex

If the index is not valid, a 0 pointer is returned.

locationCount

```
unsigned long locationCount() const;
```

Returns the number of locations stored in the exception location array.

logExceptionData

```
virtual IException &logExceptionData();
```

Logs the exception data stored in the IException object using the function specified by IException::setTraceFunction (see page 345). If you have not set a tracing function, the exception information is written to standard error output.

name

```
virtual const char *name() const;
```

Returns the name of the object's class.

setErrorId

```
IException &setErrorId(unsigned long errorId);
```

Sets the error ID to the specified value.

errorId The identifier you want to associate with this error.

setSeverity

```
IException &setSeverity(Severity severity);
```

Sets the severity of the exception.

severity Use the enumeration Severity (see page 346) to specify the severity of the exception.

setText

```
IException &setText(const char *errorText);
```

Adds the specified text to the top of the exception text stack.

errorText The error text you want to add.

setTraceFunction

```
static void setTraceFunction(IException::TraceFn &traceFunction);
```

Registers an object of IException::TraceFn (see page 357) to be used to log exception data. The ITrace (see page 579) member functions and macros write the trace messages. IException::logExceptionData (see page 345) calls IException::TraceFn::write (see page 357) 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.

IException

terminate

```
virtual void terminate();
```

Ends the application. Normally, the Application Support 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:

ITHROW	Found in IException.
IASSERTPARM	Found in IErrorInfo (see page 354)
IASSERTSTATE	Found in IBaseErrorInfo.
ITHROWLIBRARYERROR	Found in IBaseErrorInfo.
ITHROWLIBRARYERROR1	Found in IBaseErrorInfo
ITHROWGUIERROR	Found in IGUIErrorInfo (see page 359)
ITHROWGUIERROR2	Found in IGUIErrorInfo.
ITROWSYSTEMERROR	Found in ISystemErrorInfo (see page 371)

Note: OS/390 C/C++ does not support the IGUIErrorInfo class. It is listed here because versions of the Application Support Class Library on other operating systems do support it.

text

```
const char *text(unsigned long indexFromTop = 0) const;
```

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.

textCount

```
unsigned long textCount() const;
```

Returns the number of text strings in the exception text stack.

Enumerations

Severity

```
public:  
enum 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.

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.

```
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

Exception contains the following nested classes:

Exception::TraceFn (see page 357)

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.

Protected Members

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();
```

IAccessError

Derivation

```

IException
IAccessError

```

Header File

```
iexcbase.hpp
```

Objects of the IAccessError class represent an exception. When a member function makes a request of the operating system or the presentation system that cannot be satisfied, 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 (see page 369).

Note: Typically, if no other exception fits an error condition, the Application Support Class Library creates and throws an object of the IAccessError class.

Constructors

```

public:
IAccessError(
    const char *errorText, unsigned long errorId,
    Severity severity = IException::unrecoverable);

```

You can create objects of this class by doing the following:

- Using the constructor.
 - errorText* The text describing this particular error.
 - errorId* The identifier you want to associate with this particular error.
 - severity* Use the enumeration IException::Severity (see page 346) to specify the severity of the error. The default is unrecoverable.
- Using the macros discussed in IException (see page 341). The Application Support Class Library provides these macros to make creating exceptions easier for you.

Public Members

name

```
virtual const char *name() const;
```

Returns the name of the object's class.

Inherited Public Members

Member	Class	Page	Member	Class	Page
addLocation	Exception	343	name	Exception	345
appendText	Exception	344	setErrorId	Exception	345
errorId	Exception	344	setSeverity	Exception	345
isRecoverable	Exception	344	setText	Exception	345
locationAtIndex	Exception	344	terminate	Exception	346
locationCount	Exception	345	text	Exception	346
logExceptionData	Exception	345	textCount	Exception	346

IAssertionFailure

Derivation

Exception
IAssertionFailure

Header File

ixcbase.hpp

Objects of the IAssertionFailure class represent an exception. The IASSERT macro expands to create and throw an object of the IAssertionFailure class if the specified condition is not met. An assertion is a debugging tool you can use to assure a condition is true. The class Exception (see page 341) describes IASSERT and the other exception-handling macros.

Constructors

```
public:
IAssertionFailure(
    const char *errorText, unsigned long errorId,
    Severity severity = Exception::unrecoverable);
```

You can create objects of this class by doing the following:

- Using the constructor.
 - errorText* The text describing this particular error.
 - errorId* The identifier you want to associate with this particular error.
 - severity* Use the enumeration Exception::Severity (see page 346) to specify the severity of the error. The default is unrecoverable.
- Using the macro IASSERT (see page 341). The Application Support Class Library provides this macro to make creating exceptions easier for you.

Public Members

name

```
virtual const char *name() const;
```

Returns the name of the object's class.

Inherited Public Members

Member	Class	Page	Member	Class	Page
addLocation	Exception	343	name	Exception	345
appendText	Exception	344	setErrorId	Exception	345
errorId	Exception	344	setSeverity	Exception	345
isRecoverable	Exception	344	setText	Exception	345
locationAtIndex	Exception	344	terminate	Exception	346
locationCount	Exception	345	text	Exception	346
logExceptionData	Exception	345	textCount	Exception	346

ICLibErrorInfo

Derivation

```

IBase
  IBase
    IBaseErrorInfo
      ICLibErrorInfo
  
```

Header File

```
iexcept.hpp
```

Objects of the ICLibErrorInfo class represent error information. When a C library call results in an error condition, objects of the ICLibErrorInfo class are created. The per thread global variable `errno` is used to obtain the error text.

The Application Support 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` (see page 356) to specify the type of the exception. The default is `accessError`.

severity Use the enumeration `Exception::Severity` (see page 346) to specify the severity of the error. The default is `recoverable`.

This macro generates code that calls `throwCLibError` (see page 352), which does the following:

1. Creates an ICLibErrorInfo object
2. Uses the object to create an Exception object
3. Adds location information
4. Logs the exception data
5. Throws the exception

Constructors

```

public:
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 Application Support Class Library provides the following default text: "No error text is available."

CLibFunctionName

(Optional) The name of the failing C library function. If you specify *CLibFunctionName*, the constructor prefixes it to the error text.

Public Members

errorId

```
virtual unsigned long errorId() const;
```

Returns the value of errno, which you can use to obtain the errno information.

isAvailable

```
virtual Boolean isAvailable() const;
```

If the error text is available, true is returned.

operator const char *

```
virtual operator const char *() const;
```

Returns the error text.

text

```
virtual const char *text() const;
```

Returns the error text.

throwCLibError

```
static void throwCLibError(  
    const char *functionName,  
    const IExceptionLocation &location,  
    IBaseErrorInfo::ExceptionType name = accessError,  
    IException::Severity severity = recoverable);
```

Creates an ICLibErrorInfo object and uses the text from it to the following:

1. Create an exception object
2. Add the location information to it
3. Log the exception data
4. Throw the exception

functionName

The name of the function where the exception occurred.

location An IExceptionLocation (see page 358) object containing the following:

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

name Use the enumeration IBaseErrorInfo::ExceptionType (see page 356) to specify the type of the exception. The default is accessError.

severity Use the enumeration IException::Severity (see page 346) to specify the severity of the error. The default is recoverable.

Inherited Public Members

Member	Class	Page	Member	Class	Page
asDebugInfo	IBase	311	operator<<	IBase	312
asDebugInfo	IVBase	314	operator<<	IVBase	314
asString	IBase	311	operator const char*	IBaseErrorInfo	355
asString	IVBase	314	text	IBaseErrorInfo	355
errorId	IBaseErrorInfo	355	throwError	IBaseErrorInfo	356
isAvailable	IBaseErrorInfo	355			

IODeviceError

Derivation

```

IException
  IDeviceError

```

Header File

```
iexcbase.hpp
```

Objects of the IDeviceError class represent an exception. When a member function makes a hardware-related request of the operating system or the presentation system that the system cannot satisfy because of a hardware failure, the member function creates and throws an object of the IDeviceError class. An example of a failing hardware-related request is printing to a disconnected printer.

Constructors

```

public:
IODeviceError(const char *errorText, unsigned long errorId,
              Severity severity = IException::unrecoverable);

```

You can create objects of this class by doing the following:

- Using the constructor.
 - errorText* The text describing this particular error.
 - errorId* The identifier you want to associate with this particular error.
 - severity* Use the enumeration IException::Severity (see page 346) to specify the severity of the error. The default is unrecoverable.
- Using the macros discussed in IException (see page 341). The Application Support Class Library provides these macros to make creating exceptions easier.

Public Members

name

```
virtual const char *name() const;
```

Returns the name of the object's class.

Inherited Public Members

Member	Class	Page	Member	Class	Page
addLocation	IException	343	name	IException	345
appendText	IException	344	setErrorId	IException	345
errorId	IException	344	setSeverity	IException	345
isRecoverable	IException	344	setText	IException	345
locationAtIndex	IException	344	terminate	IException	346
locationCount	IException	345	text	IException	346
logExceptionData	IException	345	textCount	IException	346

IBaseErrorInfo

Derivation

```

IBase
  IVBase
    IBaseErrorInfo
  
```

Header File

```
iexcept.hpp
```

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. By using this macro, you are asserting that the expression is true. If it evaluates to false, the macro generates code that calls the IExcept__assertParameter function, which creates an IInvalidParameter (see page 362) exception. The exception data is logged using IException::TraceFn::logExceptionData, and the exception is then thrown.

IASSERTSTATE

This macro accepts an expression to test. By using this macro, you are asserting that the expression is true. If it evaluates to false, the macro generates code that calls the IExcept__assertState function, which creates an IInvalidRequest (see page 363) exception. The exception data is logged, and the exception is then thrown.

ITHROWLIBRARYERROR

This macro can throw any of the Application Support Class Library-defined exceptions.

id The ID of the message to load from the message file.

name A value from the enumeration IBaseErrorInfo::ExceptionType (see page 356), indicating the type of exception to create.

severity A value from the enumeration IException::Severity (see page 346), indicating the severity of the exception.

messageFile The name of the message file to load the exception text from.

errorGroup

The *errorGroup* associated with this error. This can be one of the values for *ErrorCodeGroup* defined in *IException* or a value you provide.

The macro generates code that calls the *IExcept__throwLibraryError* function, which does the following:

1. Loads the message text
2. Uses the message text to create an exception object
3. Adds the error group to the object
4. Adds location information
5. Logs the exception data
6. Throws the exception

ITHROWLIBRARYERROR1

This macro can throw any of the Application Support Class Library-defined exceptions. It is identical to the *ITHROWLIBRARYERROR* macro, except it has a fourth parameter:

text Replacement text for the retrieved message.

OS/2-Specific Information: *IGUIErrorInfo* (see page 359), *ISystemErrorInfo* (see page 371), and *ICLibErrorInfo* (see page 351) 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.

Public Members

errorId

```
virtual unsigned long errorId() const = 0;
```

Returns the error ID.

isAvailable

```
virtual Boolean isAvailable() const = 0;
```

If error information is available, true is returned.

operator const char *

```
virtual operator const char *() const = 0;
```

Returns the error text.

text

```
virtual const char *text() const = 0;
```

Returns the error text.

throwError

```
void throwError(
    const IExceptionLocation &location,
    IBaseErrorInfo::ExceptionType name = accessError,
    IException::Severity severity = recoverable);
```

Creates an IBaseErrorInfo object and uses it to do the following:

1. Create an exception object
2. Add the location information to the object
3. Log the exception data
4. Throw the exception

location An IExceptionLocation (see page 358) object containing the following:

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

name Use the enumeration ExceptionType (see page 356) to specify the type of the exception. The default is accessError.

severity Use the enumeration IException::Severity (see page 346) to specify the severity of the error. The default is recoverable.

Inherited Public Members

Member	Class	Page	Member	Class	Page
asDebugInfo	IBase	311	asString	IVBase	314
asDebugInfo	IVBase	314	operator<<	IBase	312
asString	IBase	311	operator<<	IVBase	314

Enumerations

ExceptionType

```
public:
enum ExceptionType
{
    accessError, deviceError,
    invalidParameter, invalidRequest,
    outOfSystemResource, outOfWindowResource,
    outOfMemory, resourceExhausted
};
```

Use these enumerators to specify the type of exception to create:

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.

Note: OS/390 C/C++ does not support the IOutOfWindowResource class. It is listed here because versions of the Application Support Class Library on other operating systems do support it.

outOfMemory

Creates an IOutOfMemory object.

resourceExhausted

Creates an IResourceExhausted object.

Exception::TraceFn

Derivation

Exception::TraceFn

Header File

isexbase.hpp

Objects of the class IException (see page 341) and its derived classes use IException::TraceFn to log exception object data.

If you want to provide your own tracing function, derive your own class from IException::TraceFn and register it with IException using IException::setTraceFunction (see page 345). You can then implement IException::TraceFn::write (see page 357), which is called during exception processing.

If you do not register an IException::TraceFn, IException uses an object of a default derived class of IException::TraceFn. This default derived class writes its output to wherever the ICLUI TRACETO environment variable directs the output from ITrace (see page 579).

If you want to completely take over exception logging you need to provide an implementation for the logData() function. This function is called by the logExceptionData() function and is passed the IException object. If you only want to process buffers of exception data in a non-default manner you should override the write() function instead. By default this function is called multiple times by the logData() function during the logging of an exception. The default behavior of the write() function is to write output to wherever ITrace output is being directed based on the ICLUI_TRACETO environment variable. An object of a default subclass of IException::TraceFn is used if your application does not register one.

Public Members

write

```
virtual void write(const char *buffer) = 0;
```

Writes a buffer of exception data. This is a pure virtual function.

ExceptionLocation

Derivation

ExceptionLocation

Header File

isexbase.hpp

Objects of the ExceptionLocation class save the location information when an exception is thrown or rethrown. 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 ExceptionLocation object when an exception is to be thrown or rethrown, respectively. However, you can create your own ExceptionLocation object by constructing it yourself or by using the IEXCEPTION_LOCATION macro.

Constructors

```
public:
ExceptionLocation(
    const char *fileName = 0, const char *functionName = 0,
    unsigned long lineNumber = 0);
```

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 (see page 341). 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.

Public Members

fileName

```
const char *fileName() const;
```

Returns the path-qualified source file name where an exception has been thrown or rethrown.

functionName

```
const char *functionName() const;
```

Returns the name of the function that has thrown or rethrown an exception.

lineNumber

```
unsigned long lineNumber() const;
```

Returns the line number of the statement in the source file from which an exception has been thrown or rethrown.

IGUIErrorInfo

Note: OS/390 C/C++ does not support the IGUIErrorInfo class. It is described here because versions of the Application Support Class Library on other operating systems do support it.

Derivation

```
IBase
  IBase
    IBaseErrorInfo
      IGUIErrorInfo
```

Header File

```
iexcept.hpp
```

Objects of the IGUIErrorInfo class represent error information that you can include in an exception object. When an OS/2 Win call results in an error condition, objects of the IGUIErrorInfo class are created. You can use the error text to construct a derived class object of IException (see page 341).

The Application Support 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 (see page 361), which does the following:

1. Creates an IGUIErrorInfo object
2. Uses the object to create an object of IAccessError (see page 349)
3. Adds location information
4. Logs the exception data
5. Throws the exception

Note: This macro uses the recoverable enumerator provided by IException::Severity (see page 346).

ITHROWGUIERROR2

This macro can throw any of the Application Support Class Library-defined exceptions. This macro accepts the following parameters:

location The name of the GUI function returning an error code, the name of the file the function is in, and the function's line number.

- name* Use the enumeration IBaseErrorInfo::ExceptionType (see page 356) to specify the type of the exception. The default is accessError.
- severity* Use the enumeration IException::Severity (see page 346) to specify the severity of the error. The default is recoverable.

This macro generates code that calls throwGUIError (see page 361), which does the following:

1. Creates an IGUIErrorInfo object
2. Uses the object to create an IException object
3. Adds location information
4. Logs the exception data
5. Throws the exception

Portability Considerations: 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.”

OS/2-Specific Information: You can use objects of the IGUIErrorInfo class to obtain information about the last error that occurred on a call to Presentation Manager.

Constructors

```
public:
IGUIErrorInfo(const char *GUIFunctionName = 0);
```

You can only construct objects of this class using the default constructor.

Note: If the constructor cannot load the error text, the Application Support Class Library provides the following default text: “No error text is available.”

GUIFunctionName

(Optional) The name of the failing GUI function. If you specify *GUIFunctionName*, the constructor prefixes it to the error text.

Public Members

errorId

```
virtual unsigned long errorId() const;
```

Returns the error ID.

OS/2-Specific Information: In the case of a Presentation Manager error, the IGUIErrorInfo constructor obtains the *errorId* using WinGetLastError.

isAvailable

```
virtual Boolean isAvailable() const;
```

If the error information is available, true is returned.

operator const char *

```
virtual operator const char *() const;
```

Returns the error text.

text

```
virtual const char *text() const;
```

Returns the error text.

throwError

```
void throwError(
    const IExceptionLocation &location,
    IBaseErrorInfo::ExceptionType name = accessError,
    IException::Severity severity = recoverable);
```

Creates an IGUIErrorInfo object and uses the text from it to do the following:

1. Create an exception object
2. Add the location information
3. Log the exception data
4. Throw the exception

location An IExceptionLocation (see page 358) object containing the following:

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

name Use the enumeration IBaseErrorInfo::ExceptionType (see page 356) to specify the type of the exception. The default is accessError.

severity Use the enumeration IException::Severity (see page 346) to specify the severity of the error. The default is recoverable.

throwGUIError

```
static void throwGUIError(
    const char *functionName,
    const IExceptionLocation &location,
    IBaseErrorInfo::ExceptionType name = accessError,
    IException::Severity severity = recoverable);
```

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.

InvalidParameter

<i>location</i>	An IExceptionLocation (see page 358) object containing the following: <ul style="list-style-type: none">• Function name• File name• Line number where the function is called
<i>name</i>	Use the enumeration IBaseErrorInfo::ExceptionType (see page 356) to specify the type of the exception. The default is accessError.
<i>severity</i>	Use the enumeration IException::Severity (see page 346) to specify the severity of the error. The default is recoverable.

Inherited Public Members

Member	Class	Page	Member	Class	Page
asDebugInfo	IBase	311	operator<<	IBase	312
asDebugInfo	IVBase	314	operator<<	IVBase	314
asString	IBase	311	operator const char*	IBaseErrorInfo	355
asString	IVBase	314	text	IBaseErrorInfo	355
errorId	IBaseErrorInfo	355	throwError	IBaseErrorInfo	356
isAvailable	IBaseErrorInfo	355			

InvalidParameter

Derivation

IException
InvalidParameter

Header File

iexcbase.hpp

Objects of the InvalidParameter class represent 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 (see page 350), with one difference: InvalidParameter is thrown whether or not you define IC_DEVELOP for the compile.

Constructors

```
public:  
InvalidParameter(  
    const char *errorText, unsigned long errorId,  
    Severity severity = IException::unrecoverable);
```

You can create objects of this class by doing the following:

- Using the constructor.

errorText The text describing this particular error.

errorId The identifier you want to associate with this particular error.

severity Use the enumeration IException::Severity (see page 346) to specify the severity of the error. The default is unrecoverable.

- Using the macros discussed in IException (see page 341). The Application

Support Class Library provides these macros to make creating exceptions easier for you.

Public Members

name

```
virtual const char *name() const;
```

Returns the name of the object's class.

Inherited Public Members

Member	Class	Page	Member	Class	Page
addLocation	Exception	343	name	Exception	345
appendText	Exception	344	setErrorId	Exception	345
errorId	Exception	344	setSeverity	Exception	345
isRecoverable	Exception	344	setText	Exception	345
locationAtIndex	Exception	344	terminate	Exception	346
locationCount	Exception	345	text	Exception	346
logExceptionData	Exception	345	textCount	Exception	346

InvalidRequest

Derivation

```
Exception
InvalidRequest
```

Header File

```
ixcbase.hpp
```

Objects of the InvalidRequest class represent 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.

Constructors

```
public:
InvalidRequest(
    const char *errorText, unsigned long errorId,
    Severity severity = Exception::unrecoverable);
```

You can create objects of this class by doing the following:

- Using the constructor.
 - errorText* The text describing this particular error.
 - errorId* The identifier you want to associate with this particular error.
 - severity* Use the enumeration Exception::Severity (see page 346) to specify the severity of the error. The default is unrecoverable.
- Using the macros discussed in Exception (see page 341). The Application Support Class Library provides these macros to make creating exceptions easier for you.

Public Members

name

```
virtual const char *name() const;
```

Returns the name of the object's class.

Inherited Public Members

Member	Class	Page	Member	Class	Page
addLocation	Exception	343	name	Exception	345
appendText	Exception	344	setErrorId	Exception	345
errorId	Exception	344	setSeverity	Exception	345
isRecoverable	Exception	344	setText	Exception	345
locationAtIndex	Exception	344	terminate	Exception	346
locationCount	Exception	345	text	Exception	346
logExceptionData	Exception	345	textCount	Exception	346

IMessageText

Derivation

IMessageText

Header File

imsgtext.hpp

Objects of the IMessageText class load message text from a message file. When the Application Support Class Library detects an error condition and prepares to throw an exception, the library creates an object of this class if you are using customized message text.

OS/2-Specific Information 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.

Constructors

```
public:
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 to insert additional text strings into the retrieved message, or the copy constructor, described below:

```
IMessageText(const IMessageText &text);
```

- Retrieving a message from a file and, optionally, inserting additional text strings within the retrieved message.

You can have 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 numbers of the text insertion parameters 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 message text cannot be retrieved from the message file, this constructor uses the following default text: "Unable to load text from message file."

OS/390-Specific Information:

On OS/390, the IMessageText class retrieves messages using the OS/390 Language Environment Messaging services. The message file name passed must be a Language Environment Message Table Module name, of the form UxxxMSGT, where xxx is a three-letter facility ID, e.g., CLB for the class libraries. See the chapter on Handling Messages in the *OS/390 Language Environment Programming Guide* for information on how to set up Language Environment Message tables and message files. The Language Environment Message Table Module must be accessible at run time, in the LPA, TSOLIB, etc. The messageId argument of the IMessageText constructor maps to the Language Environment :msgno. tag in the message source file. IMessageText does not retrieve messages coded with a :msgsubid. tag. IMessageText does not use the INCLUDE file generated by CEEBLDTX (see the *OS/390 Language Environment Programming Guide*).

textInsert1 through textInsert9

(Optional) The substitution text to be inserted in place of the substitution symbol in the message.

IOutOfMemory

- Using the Application Support Class Library provided copy constructor:
`IMessageText(const IMessageText& msgText);`
msgText The message text object you want to copy.

Public Members

operator=

`IMessageText &operator=(const IMessageText &msgText);`

Sets the object data to the values of the specified IMessageText object.

text The message text object you want to copy.

operator const char *

`operator const char *() const;`

Returns the message text.

setDefaultText

`IMessageText &setDefaultText(const char *text);`

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.

text

`const char *text() const;`

Returns the message text.

IOutOfMemory

Derivation

Exception
 IResourceExhausted
 IOutOfMemory

Header File

`iexcbase.hpp`

Objects of the IOutOfMemory class represent an exception. The Application Support Class Library's `new_handler` function creates an object of the IOutOfMemory class when heap memory is exhausted.

Constructors

```
public:
IOutOfMemory(
    const char *errorText, unsigned long errorId,
    Severity severity = IException::unrecoverable);
```

You can create objects of this class by doing the following:

- Using the constructor.

errorText The text describing this particular error.

errorId The identifier you want to associate with this particular error.

severity Use the enumeration IException::Severity (see page 346) to specify the severity of the error. The default is unrecoverable.

- Using the macros discussed in IException (see page 341). The Application Support Class Library provides these macros to make creating exceptions easier for you.

Public Members

name

```
virtual const char *name() const;
```

Returns the name of the object's class.

Inherited Public Members

Member	Class	Page	Member	Class	Page
addLocation	IException	343	name	IResource-	370
appendText	IException	344		Exhausted	
errorId	IException	344	setErrorId	IException	345
isRecoverable	IException	344	setSeverity	IException	345
locationAtIndex	IException	344	setText	IException	345
locationCount	IException	345	terminate	IException	346
logExceptionData	IException	345	text	IException	346
name	IException	345	textCount	IException	346

IOutOfSystemResource

Derivation

```
IException
IResourceExhausted
IOutOfSystemResource
```

Header File

```
isexbase.hpp
```

Objects of the IOutOfSystemResource class represent an exception. When a member function makes an operating system resource request that the system cannot satisfy, the member function creates and throws an object of the IOutOfSystemResource class.

Constructors

```
public:
IOOutOfSystemResource(
    const char *errorText, unsigned long errorId,
    Severity severity = IException::unrecoverable);
```

You can create objects of this class by doing the following:

- Using the constructor.

errorText The text describing this particular error.

errorId The identifier you want to associate with this particular error.

severity Use the enumeration IException::Severity (see page 346) to specify the severity of the error. The default is unrecoverable.

- Using the macros discussed in IException (see page 341). The Application Support Class Library provides these macros to make creating exceptions easier for you.

Public Members

name

```
virtual const char *name() const;
```

Returns the name of the object's class.

Inherited Public Members

Member	Class	Page	Member	Class	Page
addLocation	IException	343	name	IResource-	370
appendText	IException	344		Exhausted	
errorId	IException	344	setErrorId	IException	345
isRecoverable	IException	344	setSeverity	IException	345
locationAtIndex	IException	344	setText	IException	345
locationCount	IException	345	terminate	IException	346
logExceptionData	IException	345	text	IException	346
name	IException	345	textCount	IException	346

IOOutOfWindowResource

Note: OS/390 C/C++ does not support the IOOutOfWindowResource class. It is listed here because versions of the Application Support Class Library on other operating systems do support it.

Derivation

```
IException
IResourceExhausted
IOOutOfWindowResource
```

Header File

iexcbase.hpp

Objects of the IOutOfWindowResource class represent an exception. When a member function makes a resource request to a presentation (window) system resource, and the system cannot satisfy the request, the member function creates and throws an object of the IOutOfWindowResource class.

Constructors

```
public:
IOutOfWindowResource(
    const char *errorText,unsigned long errorId,
    Severity severity = IException::unrecoverable);
```

You can create objects of this class by doing the following:

- Using the constructor.
errorText The text describing this particular error.
errorId The identifier you want to associate with this particular error.
severity Use the enumeration IException::Severity (see page 346) to specify the severity of the error. The default is unrecoverable.
- Using the macros discussed in IException (see page 341). The Application Support Class Library provides these macros to make creating exceptions easier for you.

Public Members

name

```
virtual const char *name() const;
```

Returns the name of the object’s class.

Inherited Public Members

Member	Class	Page	Member	Class	Page
addLocation	IException	343	name	IResource-	370
appendText	IException	344		Exhausted	
errorId	IException	344	setErrorId	IException	345
isRecoverable	IException	344	setSeverity	IException	345
locationAtIndex	IException	344	setText	IException	345
locationCount	IException	345	terminate	IException	346
logExceptionData	IException	345	text	IException	346
name	IException	345	textCount	IException	346

IResourceExhausted

IResourceExhausted

Derivation

Exception
IResourceExhausted

Header File

isexbase.hpp

Objects of the IResourceExhausted class represent an exception. When a member function makes a resource request of the operating system or the presentation system that it cannot satisfy, the member function creates and throws an object of the IResourceExhausted class or one of its derived classes. IResourceExhausted is the generic out-of-resource class. Member functions use IResourceExhausted whenever its derived classes, which are for specific out-of-resource cases, are not applicable.

The derived classes for IResourceExhausted are:

IOutOfMemory (see page 366)
IOutOfSystemResource (see page 367)
IOutOfWindowResource (see page 368).

Note: OS/390 C/C++ does not support the IOutOfWindowResource class. It is listed here because versions of the Application Support Class Library on other operating systems do support it.

Constructors

```
public:  
IResourceExhausted(  
    const char *errorText, unsigned long errorId,  
    Severity severity = IException::unrecoverable);
```

You can create objects of this class by doing the following:

- Using the constructor.
errorText The text describing this particular error.
errorId The identifier you want to associate with this particular error.
severity Use the enumeration IException::Severity (see page 346) to specify the severity of the error. The default is unrecoverable.
- Using the macros discussed in IException (see page 341). The Application Support Class Library provides these macros to make creating exceptions easier for you.

Public Members

name

```
virtual const char *name() const;
```

Returns the name of the object's class.

Inherited Public Members

Member	Class	Page	Member	Class	Page
addLocation	IException	343	name	IException	345
appendText	IException	344	setErrorId	IException	345
errorId	IException	344	setSeverity	IException	345
isRecoverable	IException	344	setText	IException	345
locationAtIndex	IException	344	terminate	IException	346
locationCount	IException	345	text	IException	346
logExceptionData	IException	345	textCount	IException	346

ISystemErrorInfo

Derivation

IBase
IVBase
IBaseErrorInfo
ISystemErrorInfo

Header File

except.hpp

Objects of the ISystemErrorInfo class represent error information that you can include in an exception object. When an OS/2 DOS system call results in an error condition, objects of the ISystemErrorInfo class are created. You can use the error text to construct a derived class object of IException (see page 341).

The Application Support 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 (see page 356), which specifies the type of exception this macro creates
- One of the values of the enumeration IException::Severity (see page 346), which specifies the severity of the exception

This macro generates code that calls throwSystemError (see page 372), which does the following:

1. Creates an ISystemErrorInfo object
2. Uses the object to create an IException object
3. Adds location information
4. Logs the exception data
5. Throws the exception

OS/390 C++-Specific Information: You can create objects of this class on OS/390 C/C++, but the objects contain no useful information and only have the default message: "System exception condition detected."

Constructors

```
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 Application Support Class Library provides the following default text: “No error text is available.”

systemErrorId

The error ID identifying an operating system error.

systemFunctionName

(Optional) The name of the failing system call that returned the error ID. If you specify *systemFunctionName*, the constructor prefixes it to the error text.

Public Members

errorId

```
virtual unsigned long errorId() const;
```

Returns the error ID.

isAvailable

```
virtual Boolean isAvailable() const;
```

If the error information is available, true is returned.

operator const char *

```
virtual operator const char *() const;
```

Returns the error text.

text

```
virtual const char *text() const;
```

Returns the error text.

throwSystemError

```
static void throwSystemError(  
    unsigned long systemErrorId,  
    const char *functionName,  
    const IExceptionLocation &location,  
    IBaseErrorInfo::ExceptionType name = accessError  
    IException::Severity severity = recoverable);
```

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 (see page 358) object containing the following:

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

name Use the enumeration IBaseErrorInfo::ExceptionType (see page 356) to specify the type of the exception. The default is accessError.

severity Use the enumeration IException::Severity (see page 346) to specify the severity of the error. The default is recoverable.

Inherited Public Members

Member	Class	Page	Member	Class	Page
asDebugInfo	IBase	311	operator<<	IBase	312
asDebugInfo	IVBase	314	operator<<	IVBase	314
asString	IBase	311	operator const char*	IBaseErrorInfo	355
asString	IVBase	314	text	IBaseErrorInfo	355
errorId	IBaseErrorInfo	355	throwError	IBaseErrorInfo	356
isAvailable	IBaseErrorInfo	355			

IXLibErrorInfo

Note: The IXLibErrorInfo class is provided for versions of the Application Support Class Library that support the User Interface Class Library and that run on X/Windows**-based windowing systems. In the OS/2, OS/390, and AS/400 versions of the library, this class is not supported.

Derivation

```

IBase
  IVBase
    IBaseErrorInfo
      IXLibErrorInfo
  
```

Header File

iexcept.hpp

Objects of the IXLibErrorInfo class represent error information that you can include in an exception object. When an X library call results in an error condition, objects of the IXLibErrorInfo class are created. IThread registers a handler through XSetErrorHandler to do the following:

- Detect the error condition
- Save the error code

You can use this error code to obtain the information about the X library error. When you have an X library function call fail, construct an object of this class to obtain the error text. You can use the error text to construct a derived class object of IException (see page 341).

The Application Support 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 (see page 356) to specify the type of the exception. The default is accessError.
- severity* Use the enumeration IException::Severity (see page 346) to specify the severity of the error. The default is recoverable.

This macro generates code that calls throwXLibError (see page 375), 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

Constructors

```
public:  
IXLibErrorInfo(const char *systemFunctionName = 0);
```

Note: This member is available on X/Windows-based systems only.

You can only construct objects of this class using the default constructor.

Note: If the constructor cannot load the error text, the Application Support Class Library provides the following default text: “No error text is available.”

systemFunctionName
(Optional) The name of the failing X library function. If you specify *systemFunctionName*, the constructor prefixes it to the error text.

Public Members

errorId

```
virtual unsigned long errorId() const;
```

Returns the X error code, which you can use to obtain the error text.

Note: This member is available on X/Windows-based systems only.

isAvailable

```
virtual Boolean isAvailable() const;
```

If the error text is available, true is returned.

Note: This member is available on X/Windows-based systems only.

operator const char *

```
virtual operator const char *() const;
```

Returns the error text.

Note: This member is available on X/Windows-based systems only.

text

```
virtual const char *text() const;
```

Returns the error text.

Note: This member is available on X/Windows-based systems only.

throwXLibError

```
static void throwXLibError(
    const char *functionName,
    const IExceptionLocation &location,
    IBaseErrorInfo::ExceptionType name = accessError,
    IException::Severity severity = recoverable);
```

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

Note: This member is available on X/Windows-based systems only.

functionName

The name of the function where the exception occurred.

location An IExceptionLocation (see page 358) object containing the following:

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

name Use the enumeration IBaseErrorInfo::ExceptionType (see page 356) to specify the type of the exception. The default is accessError.

severity Use the enumeration IException::Severity (see page 346) to specify the severity of the error. The default is recoverable.

Inherited Public Members

Member	Class	Page	Member	Class	Page
asDebugInfo	IBase	311	operator<<	IBase	312
asDebugInfo	IVBase	314	operator<<	IVBase	314
asString	IBase	311	operator const char*	IBaseErrorInfo	355
asString	IVBase	314	text	IBaseErrorInfo	355
errorId	IBaseErrorInfo	355	throwError	IBaseErrorInfo	355
isAvailable	IBaseErrorInfo	355			

Chapter 60. String Classes

IString

Derivation

```
IBase
IString
```

Header File

```
istring.hpp
```

Objects of the IString class are arrays of characters. These objects are functionally equivalent to objects of the class I0String (see page 405) with one major distinction: IStrings are indexed starting at 1 instead of 0. (An index value of zero used in an IString member function is automatically converted to 1 when the function is called.)

IString provides an operator `char*`. To access the actual string contained in an object of type IString, cast the assignment variable implicitly or explicitly.

IStrings provide the following functions beyond that available from the standard C `char*` arrays and the `string.h` library functions:

- No restrictions on string contents. Thus, strings can contain null characters.
- Automatic conversion from and to numeric types.
- Automatic deletion of the string buffer when the IString is destroyed.
- Full support for the following:
 - All comparison operators
 - All bitwise operators
 - Concatenation using the more natural `+` operator
- String data testing, such as for characters, digits, and hexadecimal digits.
- A full complement of the following:
 - String manipulation functions, such as center, left- and right-justification, stripping of leading and trailing characters, deleting substrings, and inserting strings
 - Corresponding string manipulation functions that return a new IString rather than modifying the receiver
 - String searching functions, such as byte index of string and last-byte index of string.
- Word manipulation, such as index of word and search for word phrase.
- Support for mixed strings that contain both single-byte character set (SBCS) and double-byte character set (DBCS) characters.

When a program using IStrings is run on a DBCS system, the IString objects support DBCS characters within the string contents. The various IString search functions do not accidentally match an SBCS character with the second byte of a DBCS character that has the same value. Also, IString functions that modify

IStrings, such as `subString` (see page 399), `remove` (see page 396), and `translate` (see page 399), never separate the two bytes of a DBCS character. If one of the two bytes of a DBCS character is removed, the remaining byte is replaced with the appropriate pad character (if the function performing the change has one) or a blank.

When working with IStrings that contain DBCS data, ensure that the contents are not altered in such a way as to corrupt the data. For example, the statement:

```
aString[ n ] = 'x';
```

would be in error if the `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`.

IStrings are held in IBuffers that allocate the area for the character arrays using the C++ operator `new`. The only limitations for the size of an IString are those imposed by the operating system.

OS/390 C++-Specific Information: DBCS is equivalent to multiple-byte character set (MBCS).

Constructors

```
public:
IString();
IString(const IString &);
IString(const IString *);
IString(int);
IString(unsigned);
IString(long);
IString(unsigned long);
IString(short);
IString(unsigned short);
IString(long long);
IString(unsigned long long);
IString(double);
IString(char);
IString(unsigned char);
IString(signed char);
IString(const char *);
IString(const unsigned char *);
IString(const signed char *);
IString(const void *pBuffer1, unsigned lenBuffer1,
        char padCharacter = ' ');
IString(const void *pBuffer1, unsigned lenBuffer1,
        const void *pBuffer2, unsigned lenBuffer2,
        char padCharacter = ' ');
IString(const void *pBuffer1, unsigned lenBuffer1,
        const void *pBuffer2, unsigned lenBuffer2,
        const void *pBuffer3, unsigned lenBuffer3,
        char padCharacter = ' ');
```

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 integer and double types.

- 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, a zero value can be specified for the void*; in this case, an IString is created of the specified length and the contents are initialized to a given pad character. The default pad character is a blank.

The pad character is only used when the initial parameter is zero. If the specified buffer has a different length than the length specified in the second parameter, no padding is done; instead, the second parameter is ignored and the buffer length is used.

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 Application Support Class Library translates memory allocation errors into exceptions. Generally, such errors do not occur until you allocate a large amount of storage, for example, more than your system's virtual memory can accommodate.

- Out-of-range errors

These occur if you attempt to construct an IString with a length greater than `UINT_MAX`.

Public Members

asDebugInfo

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

Returns information about the IString's internal representation that you can use for debugging.

asDouble

```
double asDouble() const;
```

Returns, as a double, the number that the string represents.

asInt

```
long asInt() const;
```

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

asLongLong

```
long long asLongLong() const;
```

Returns, as a long long, the number that the string represents.

asString

```
IString asString() const;
```

Returns the string itself, so that IString supports this common IBase (see page 311) protocol.

asUnsigned

```
unsigned long asUnsigned() const;
```

Returns, as an unsigned long, the integer that the string represents.

asUnsignedLongLong

```
unsigned long long asUnsignedLongLong() const;
```

Returns, as an unsigned long long, the number that the string represents.

b2c

```
static IString b2c(const IString &aString);  
IString &b2c();
```

Converts a string of binary digits to a normal string of characters. For example, this function changes 01 to \x01 and 11110011 to 3.

Note: This function is not locale-sensitive; mappings are based on code page IBM-1047 only.

b2d

```
IString &b2d();  
static IString b2d(const IString &aString);
```

Converts a string of binary digits to a string of decimal digits. For example, this function changes 00011001 to 25 and 0001001000110100 to 4660.

b2x

```
static IString b2x(const IString &aString);  
IString &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.

c2b

```
IString &c2b();  
static IString c2b(const IString &aString);
```

Converts a normal string of characters to a string of binary digits. For example, this function changes "a" to 10000001 and 12 to 1111000111110010.

Note: This function is not locale-sensitive; mappings are based on code page IBM-1047 only.

c2d

```
IString &c2d();
static IString c2d(const IString &aString);
```

Converts a normal string of characters to a string of decimal digits. For example, this function changes “a” to 129 and “ab” to 33154.

Note: This function is not locale-sensitive; mappings are based on code page IBM-1047 only.

c2x

```
IString &c2x();
static IString c2x(const IString &aString);
```

Converts a normal string of characters to a string of hexadecimal digits. For example, this function changes “a” to 81 and “ab” to 8182.

Note: This function is not locale-sensitive; mappings are based on code page IBM-1047 only.

center

```
static IString center(const IString &aString, unsigned length,
                     char padCharacter = ' ');
```

```
IString &center(unsigned length, char padCharacter = ' ');
```

Centers the receiver within a string of the specified length.

change

```
static IString change(const IString &aString, const char *pInputString,
                    const char *pOutputString, unsigned startPos = 1,
                    unsigned numChanges = ( unsigned ) UINT_MAX);
```

```
IString &change(const IString &inputString, const IString &outputString,
               unsigned startPos = 1, unsigned numChanges = ( unsigned ) UINT_MAX);
```

```
IString &change(const IString &inputString, const char *pOutputString,
               unsigned startPos = 1, unsigned numChanges = ( unsigned ) UINT_MAX);
```

```
IString &change(const char *pInputString, const IString &outputString,
               unsigned startPos = 1, unsigned numChanges = ( unsigned ) UINT_MAX);
```

```
IString &change(const char *pInputString, const char *pOutputString,
               unsigned startPos = 1, unsigned numChanges = ( unsigned ) UINT_MAX);
```

```
static IString change(const IString &aString, const IString &inputString,
                    const IString &outputString, unsigned startPos = 1,
                    unsigned numChanges = ( unsigned ) UINT_MAX);
```

```
static IString change(const IString &aString, const IString &inputString,
                    const char *pOutputString, unsigned startPos = 1,
                    unsigned numChanges = ( unsigned ) UINT_MAX);
```

```
static IString change(const IString &aString, const char *pInputString,
                    const IString &outputString, unsigned startPos = 1,
                    unsigned numChanges = ( unsigned ) UINT_MAX);
```

```
protected:
```

```
IString &change(const char *pPattern, unsigned patternLen,
               const char *pReplacement, unsigned replacementLen,
               unsigned startPos, unsigned numChanges);
```

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:

<i>inputString</i>	The pattern string as a reference to an object of type IString. The library searches for the pattern string within the receiver's data.
<i>pInputString</i>	The pattern string as a null-terminated string. The library searches for the pattern string within the receiver's data.
<i>outputString</i>	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.
<i>pOutputString</i>	The replacement string as a null-terminated string. It replaces the occurrences of the pattern string in the receiver's data.
<i>startPos</i>	The position to start the search at within the receiver's data. The default is 1.
<i>numChanges</i>	The number of patterns to search for and change. The default is <code>UINT_MAX</code> , which causes changes to all occurrences of the pattern.

charType

```
IStringEnum::CharType charType(unsigned index) const;
```

Returns the type of the character at the specified index.

copy

```
static IString copy(const IString &aString, unsigned numCopies);  
IString &copy(unsigned numCopies);
```

Replaces the receiver's contents with a specified number of replications of itself.

d2b

```
static IString d2b(const IString &aString);  
IString &d2b();
```

Converts a string of decimal digits to a string of binary digits. For example, this function changes 12 to 00001100 and 123 to 01111011. Converted values are padded on the left with zeros until the number of binary digits is a multiple of 8. If you want to strip leading zeros, you can use the `stripLeading()` function with a `char` argument:

```
IString decimal="123";  
cout << decimal.d2b() << '\n'  
    << decimal.d2b().stripLeading('0') << endl;
```

This example prints the values 01111011 and 1111011 on separate lines.

d2c

```
static IString d2c(const IString &aString);  
IString &d2c();
```

Converts a string of decimal digits to a normal string of characters. For example, this function changes 12 to `\x0c` and 248 to 8.

Note: This function is not locale-sensitive; mappings are based on code page IBM-1047 only.

d2x

```
static IString d2x(const IString &aString);
IString &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.”

disableInternationalization

```
static void
disableInternationalization();
```

Disables locale-based string operations.

enableInternationalization

```
static void
enableInternationalization( Boolean enable = true );
```

Enables locale-based string operations.

includes

```
Boolean includes(char aChar) const;
Boolean includes(const IString &aString) const;
Boolean includes(const char *pString) const;
Boolean includes(const IStringTest &aTest) const;
```

If the receiver contains the specified search string, true is returned.

includesDBCS

```
Boolean includesDBCS() const;
```

If any characters are DBCS (double-byte character set), true is returned.

Note: This function is interchangeable with includesMBCS.

includesMBCS

```
Boolean includesMBCS() const;
```

If any characters are MBCS (multiple-byte character set), true is returned.

Note: This function is interchangeable with includesDBCS.

includesSBCS

```
Boolean includesSBCS() const;
```

If any characters are SBCS (single-byte character set), true is returned.

indexOf

```
unsigned indexOf(const IStringTest &aTest, unsigned startPos = 1) const;
unsigned indexOf(const IString &aString, unsigned startPos = 1) const;
unsigned indexOf(const char *pString, unsigned startPos = 1) const;
unsigned indexOf(char aCharacter, unsigned startPos = 1) const;
```

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 (see page 417).

indexOfAnyBut

```
unsigned indexOfAnyBut(const IString &validChars, unsigned startPos = 1) const;  
unsigned indexOfAnyBut(const char *pValidChars, unsigned startPos = 1) const;  
unsigned indexOfAnyBut(char validChar, unsigned startPos = 1) const;  
unsigned indexOfAnyBut(const IStringTest &aTest, unsigned startPos = 1) const;
```

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 (see page 417) object.

indexOfAnyOf

```
unsigned indexOfAnyOf(char searchChar, unsigned startPos = 1) const;  
unsigned indexOfAnyOf(const IString &searchChars, unsigned startPos = 1) const;  
unsigned indexOfAnyOf(const char *pSearchChars, unsigned startPos = 1) const;  
unsigned indexOfAnyOf(const IStringTest &aTest, unsigned startPos = 1) const;
```

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 (see page 417) object.

indexOfPhrase

```
unsigned indexOfPhrase(const IString &wordString, unsigned startWord = 1) const;
```

Returns the position of the first occurrence of the specified phrase in the receiver. If the phrase is not found, 0 is returned.

indexOfWord

```
unsigned indexOfWord(unsigned wordNumber) const;  
protected:  
unsigned indexOfWord(unsigned wordNumber, unsigned startPos,  
                    unsigned numWords) const;
```

Returns the index of the specified white-space-delimited word in the receiver. If the word is not found, 0 is returned.

insert

```
IString &insert(const IString &aString, unsigned index = 0,  
              char padCharacter = ' ');  
  
IString &insert(const char *pString, unsigned index = 0,  
              char padCharacter = ' ');  
  
static IString insert(const IString &aString, const IString &anInsert,  
                    unsigned index = 0, char padCharacter = ' ');  
  
static IString insert(const IString &aString, const char *pInsert,  
                    unsigned index = 0, char padCharacter = ' ');  
  
protected:  
IString &insert(const char *pInsert, unsigned insertLen,  
              unsigned startPos, char padCharacter);
```

Inserts the specified string after the specified location.

isAbbreviationFor

```
Boolean isAbbreviationFor(
    const char *pFullString,
    unsigned minAbbrevLength = 0) const;
```

```
Boolean isAbbreviationFor(
    const IString &fullString,
    unsigned minAbbrevLength = 0) const;
```

If the receiver is a valid abbreviation of the specified string, true is returned. A value of false is returned for null IString.

The parameters are the following:

fullString The full string for the abbreviation check is contained in another IString.

pFullString The full string for the abbreviation check is a null-terminated character string.

minAbbrevLength The minimum length to match for it to be a valid abbreviation. The default minimum length is 0, which means the minimum length is the length of the receiver's string.

isAlphabetic

```
Boolean isAlphabetic() const;
```

If all the characters are in {'A'-'Z','a'-'z'}, true is returned. A value of false is returned for null IString.

Platform-Specific Information: Alphabetic characters are determined using the `isalpha()` function defined in the `cntrl` locale source file and in the `alpha` class of the `LC_CTYPE` category of the current locale.

isAlphanumeric

```
Boolean isAlphanumeric() const;
```

If all the characters are in {'A'-'Z','a'-'z','0'-'9'}, true is returned. A value of false is returned for null IString.

Platform-Specific Information: Alphanumeric characters are determined using the `isalnum()` function defined in the `cntrl` locale source file and in the `alnum` class of the `LC_CTYPE` category of the current locale.

isASCII

```
Boolean isASCII() const;
```

Returns true if the characters are in {0x00-0x7F}. A value of false is returned for null IString.

isBinaryDigits

```
Boolean isBinaryDigits() const;
```

Returns true if the characters are either 0 or 1. A value of false is returned for null IString.

IString

isControl

```
Boolean isControl() const;
```

Returns true if all the characters are control characters. A value of false is returned for null IString.

Platform-Specific Information: Control characters are determined using the `isctrl()` 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}.

isDBCS

```
Boolean isDBCS() const;
```

If all the characters are DBCS, true is returned. A value of false is returned for null IString.

Note: This function is interchangeable with `isMBCS`.

isDigits

```
Boolean isDigits() const;
```

If all the characters are in {'0'-'9'}, true is returned. A value of false is returned for null IString.

isGraphics

```
Boolean isGraphics() const;
```

Returns true if all the characters are graphics characters. A value of false is returned for null IString.

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}.

isHexDigits

```
Boolean isHexDigits() const;
```

If all the characters are in {'0'-'9','A'-'F','a'-'f'}, true is returned. A value of false is returned for null IString.

isInternationalized

```
static Boolean  
isInternationalized();
```

Determines if locale-based string operation is in effect. A value of false is returned for null IString.

isLike

```
Boolean isLike(const char *pPattern, char zeroOrMore = ' * ',  
               char anyChar = '?') const;
```

```
Boolean isLike(const IString &aPattern, char zeroOrMore = ' * ',  
               char anyChar = '?') const;
```

protected:

```
Boolean isLike(const char *pPattern, unsigned patternLen,
```



```
char zeroOrMore, char anyChar) const;
```

If the receiver matches the specified pattern, which can contain wildcard characters, true is returned. A value of false is returned for null IString.

- You can use the first wildcard character to specify that 0 or more arbitrary characters are accepted. The default wildcard character that does this is *, but you can specify another character when calling IString::isLike. For example:

```
IString( "Allison" ).isLike( "Al*ison" ) -> true
```

- You can use the second wildcard character to specify that a single arbitrary character is accepted. The default wildcard character that does this is ?, but you can specify another character when calling IString::isLike. For example:

```
IString( "istring7.cpp" ).isLike( "i*.?pp" ) -> true
IString( "Not a question!" ).isLike( "*?", '*', '-' ) -> false
```

isLowerCase

```
Boolean isLowerCase() const;
```

If all the characters are in {'a'-'z'}, true is returned. A value of false is returned for null IString.

isMBCS

```
Boolean isMBCS() const;
```

If all the characters are MBCS, true is returned. A value of false is returned for null IString.

Note: This function is interchangeable with isDBCS.

isPrintable

```
Boolean isPrintable() const;
```

Returns true if all the characters are printable characters. A value of false is returned for null IString.

Printable characters are defined by the isprint() 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}.

isPunctuation

```
Boolean isPunctuation() const;
```

If none of the characters is white space, a control character, or an alphanumeric character, true is returned. A value of false is returned for null IString.

isSBCS

```
Boolean isSBCS() const;
```

If all the characters are SBCS, true is returned. A value of false is returned for null IString.

IString

isUpperCase

```
Boolean isUpperCase() const;
```

If all the characters are in {'A'-'Z'}, true is returned. A value of false is returned for null IString.

isValidDBCS

```
Boolean isValidDBCS() const;
```

If no DBCS characters have a 0 second byte, true is returned. A value of false is returned for null IString.

Note: This function is interchangeable with isValidMBCS.

isValidMBCS

```
Boolean isValidMBCS() const;
```

If no MBCS characters have a 0 second byte, true is returned. A value of false is returned for null IString.

Note: This function is interchangeable with isValidDBCS.

isWhiteSpace

```
Boolean isWhiteSpace() const;
```

Returns true if all the characters are whitespace characters. A value of false is returned for null IString.

Whitespace characters are defined by the isspace() function as defined in the space locale source file and in the space class of the LC_CTYPE category of the current locale. For example, on ASCII systems, printable characters are those in the range {0x09-0x0D,0x20}.

lastIndexOf

```
unsigned lastIndexOf(const IString &aString,  
                    unsigned startPos = ( unsigned ) UINT_MAX) const;
```

```
unsigned lastIndexOf(const char *pString,  
                    unsigned startPos = ( unsigned ) UINT_MAX) const;
```

```
unsigned lastIndexOf(char aCharacter,  
                    unsigned startPos = ( unsigned ) UINT_MAX) const;
```

```
unsigned lastIndexOf(const IStringTest &aTest,  
                    unsigned startPos = ( unsigned ) UINT_MAX) const;
```

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 $0 \leq x \leq \text{startPos}$. 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, the search target must occur at the beginning of the string.

lastIndexOfAnyBut

```

unsigned lastIndexOfAnyBut(char validChar,
                           unsigned startPos = ( unsigned ) UINT_MAX) const;

unsigned lastIndexOfAnyBut(const IString &validChars,
                           unsigned startPos = ( unsigned ) UINT_MAX) const;

unsigned lastIndexOfAnyBut(const char *pValidChars,
                           unsigned startPos = ( unsigned ) UINT_MAX) const;

unsigned lastIndexOfAnyBut(const IStringTest &aTest,
                           unsigned startPos = ( unsigned ) UINT_MAX) const;

```

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, the search target must occur at the beginning of the string.

lastIndexOfAnyOf

```

unsigned lastIndexOfAnyOf(const IStringTest &aTest,
                           unsigned startPos = ( unsigned ) UINT_MAX) const;

unsigned lastIndexOfAnyOf(const IString &searchChars,
                           unsigned startPos = ( unsigned ) UINT_MAX) const;

unsigned lastIndexOfAnyOf(const char *pSearchChars,
                           unsigned startPos = ( unsigned ) UINT_MAX) const;

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

```

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, the search target must occur at the beginning of the string.

leftJustify

```

static IString leftJustify(const IString &aString,
                           unsigned length, char padCharacter = ' ');

IString &leftJustify(unsigned length, char padCharacter = ' ');

```

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.

length

```

unsigned length() const;

```

Returns the length of the string, not counting the terminating null character.

IString

lengthOfWord

```
unsigned lengthOfWord(unsigned wordNumber) const;
```

Returns the length of the specified white-space-delimited word in the receiver.

lineFrom

```
static IString lineFrom(istream &aStream, char delim = '\n');
```

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.

lowerCase

```
IString &lowerCase();
```

```
static IString lowerCase(const IString &aString);
```

Translates all uppercase letters in the receiver to lowercase.

numWords

```
unsigned numWords() const;
```

Returns the number of words in the receiver.

occurrencesOf

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

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

```
unsigned occurrencesOf(const char *pString, unsigned startPos = 1) const;
```

```
unsigned occurrencesOf(char aCharacter, unsigned startPos = 1) const;
```

```
protected:
```

```
unsigned occurrencesOf(const char *pSearchString, unsigned searchLen,  
                        unsigned startPos) const;
```

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 (see page 383).

operator!=

```
friend Boolean operator!=(const char *pString1, const IString &string2);  
Boolean operator!=(const IString &string1, const IString &string2);  
Boolean operator!=(const IString &string1, const char *pString2);  
Boolean operator!=(const char *pString1, const IString &string2);  
friend Boolean operator!=(const IString &string1, const IString &string2);  
friend Boolean operator!=(const IString &string1, const char *pString2);
```

If the strings are not bitwise identical, true is returned. This function can handle three forms of the comparison:

string1!=string2

Both operands are of type IString.

string1!=pString2

The first operand is an IString, and the second is a null-terminated character string.

pString1!=string2

The first operand is a null-terminated character string, and the second is an IString.

operator&

```
IString operator&(const IString &aString) const;
IString operator&(const char *pString) const;
IString operator&(const char *pString, const IString &aString);
friend IString operator&(const char *pString, const IString &aString);
```

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.

operator&=

```
IString &operator&=(const char *pString);
IString &operator&=(const IString &aString);
```

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.

operator+

```
IString operator+(const IString &aString) const;
IString operator+(const char *pString) const;
friend IString operator+(const char *pString, const IString &aString);
```

Concatenates two strings. This function can handle the following three forms:

string1 + aString

Both operands are of type IString.

string1 + pString

The first operand is an IString, and the second is a null-terminated character string.

pString + aString

The first operand is a null-terminated character string, and the second is an IString.

operator+=

```
IString &operator+=(const IString &aString);  
IString &operator+=(const char *pString);
```

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.

operator<

```
friend Boolean operator<(const IString &string1, const IString &string2);  
Boolean operator<(const IString &string1, const IString &string2);  
Boolean operator<(const IString &string1, const char *pString2);  
Boolean operator<(const char *pString1, const IString &string2);  
friend Boolean operator<(const IString &string1, const char *pString2);  
friend Boolean operator<(const char *pString1, const IString &string2);
```

If the first string is less than the second, applying the standard collating scheme (memcmp), true is returned. This function can handle three forms of the comparison:

string1 < string2

Both operands are of type IString.

string1 < pString2

The first operand is an IString, and the second is a null-terminated character string.

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<<

```
friend ostream &operator<<(ostream &aStream, const IString &aString);
```

Puts an IString's contents to an output stream.

operator<=

```
friend Boolean operator<=(const char *pString1, const IString &string2);  
Boolean operator<=(const IString &string1, const IString &string2);  
Boolean operator<=(const IString &string1, const char *pString2);  
Boolean operator<=(const char *pString1, const IString &string2);  
friend Boolean operator<=(const IString &string1, const IString &string2);  
friend Boolean operator<=(const IString &string1, const char *pString2);
```

Equivalent to (string1 < string2) || (string1 == string2). This function can handle three forms of the comparison:

string1 <= string2

Both operands are of type IString.

string1 <= pString2

The first operand is an IString, and the second is a null-terminated character string.

pString1 <= string2

The first operand is a null-terminated character string, and the second is an IString.

Note: This operator is not locale-sensitive; mappings are based on code page IBM-1047 only. 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=

```
IString &operator=(const IString &aString);
```

Replaces the contents of the string.

operator==

```
friend Boolean operator==(const IString &string1, const char *pString2);
Boolean operator==(const IString &string1, const char *pString2);
Boolean operator==(const char *pString1, const IString &string2);
friend Boolean operator==(const IString &string1, const IString &string2);
friend Boolean operator==(const char *pString1, const IString &string2);
```

If the strings are bitwise identical, true is returned. This function can handle three forms of the comparison:

string1 == string2

Both operands are of type IString.

string1 == pString2

The first operand is an IString, and the second is a null-terminated character string.

pString1 == string2

The first operand is a null-terminated character string, and the second is an IString.

operator>

```
friend Boolean operator>(const IString &string1, const IString &string2);
Boolean operator>(const IString &string1, const IString &string2);
Boolean operator>(const IString &string1, const char *pString2);
Boolean operator>(const char *pString1, const IString &string2);
friend Boolean operator>(const IString &string1, const char *pString2);
friend Boolean operator>(const char *pString1, const IString &string2);
```

Equivalent to !(string1 <= string2). This function can handle three forms of the comparison:

string1 > string2

Both operands are of type IString.

string1 > pString2

The first operand is an IString, and the second is a null-terminated character string.

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);
Boolean operator>=(const IString &string1, const char *pString2);
Boolean operator>=(const char *pString1, const IString &string2);
friend Boolean operator>=(const IString &string1, const IString &string2);
friend Boolean operator>=(const IString &string1, const char *pString2);
friend Boolean operator>=(const char *pString1, const IString &string2);
```

Equivalent to `!(string1 < string2)`. This function can handle three forms of the comparison:

string1 >= string2

Both operands are of type `IString`.

string1 >= pString2

The first operand is an `IString`, and the second is a null-terminated character string.

pString1 >= string2

The first operand is a null-terminated character string, and the second is an `IString`.

Note: This operator is not locale-sensitive; mappings are based on code page IBM-1047 only. 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>>

```
friend istream &operator>>(istream &aStream, IString &aString);
```

Puts the next white-space-delimited word from an input stream into an `IString`.

operator char *

```
operator char *() const;
```

Returns a `char*` pointer to the string's contents.

operator signed char *

```
operator signed char *() const;
```

Returns a signed `char*` pointer to the string's contents.

operator unsigned char *

```
operator unsigned char *() const;
```

Returns an unsigned `char*` pointer to the string's contents.

operator[]

```
const char &operator[](unsigned index) const;
char &operator[](unsigned index);
```

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.

operator^

```
IString operator^(const char *pString, const IString &aString);
IString operator^(const IString &aString) const;
IString operator^(const char *pString) const;
friend IString operator^(const char *pString, const IString &aString);
```

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.

operator^=

```
IString &operator^=(const char *pString);
IString &operator^=(const IString &aString);
```

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.

operator|

```
IString operator|(const IString &aString) const;
IString operator|(const char *pString) const;
IString operator|(const char *pString, const IString &aString);
friend IString operator|(const char *pString, const IString &aString);
```

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.

operator|=

```
IString &operator|=(const char *pString);
IString &operator|=(const IString &aString);
```

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.

operator~

```
IString operator~() const;
```

Returns the string's bitwise negation (the string's complement).

overlayWith

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

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

```
IString &overlayWith(
    const char *pString, unsigned index = 1,
    char padCharacter = ' ');
```

```
static IString overlayWith(
    const IString &aString, const char *pOverlay,
    unsigned index = 1, char padCharacter = ' ');
```

protected:

```
IString &overlayWith(
    const char *pOverlay, unsigned overlayLen,
    unsigned index, char padCharacter);
```

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*).

remove

```
static IString remove(const IString &aString,
    unsigned startPos, unsigned numChars);
IString &remove(unsigned startPos);
IString &remove(unsigned startPos, unsigned numChars);
static IString remove(const IString &aString, unsigned startPos);
```

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.

removeWords

```
static IString removeWords(const IString &aString, unsigned startWord);
IString &removeWords(unsigned firstWord);
IString &removeWords(unsigned firstWord, unsigned numWords);
static IString removeWords(const IString &aString,
                           unsigned startWord, unsigned numWords);
```

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 number of words defaults to the rest of the string.

Note: The static functions `IString::space` (see page 397) and `IString::removeWords` obtain the same result but do not affect the String to which they are applied.

reverse

```
static IString reverse(const IString &aString);
IString &reverse();
```

Reverses the receiver's contents.

rightJustify

```
static IString rightJustify(const IString &aString,
                           unsigned length, char padCharacter = ' ');
IString &rightJustify(unsigned length, char padCharacter = ' ');
```

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.

size

```
unsigned size() const;
```

Returns the length of the string, not counting the terminating null character.

space

```
static IString space(const IString &aString,
                    unsigned numSpaces = 1, char spaceChar = ' ');
IString &space(unsigned numSpaces = 1, char spaceChar = ' ');
```

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` (see page 397) obtain the same result but do not affect the String to which they are applied.

strip

```
IString &strip(const IString &aString);
IString &strip();
IString &strip(char aCharacter);
IString &strip(const char *pString);
IString &strip(const IStringTest &aTest);
static IString strip(const IString &aString, char aChar);
static IString strip(const IString &aString, const IString &aStringOfChars);
static IString strip(const IString &aString, const char *pStringOfChars);
static IString strip(const IString &aString, const IStringTest &aTest);

protected:
IString &strip(const char *p, unsigned len, IStringEnum::StripMode mode);
```

```
protected:  
IString &strip(const IStringTest &aTest, IStringEnum::StripMode mode);
```

Strips both leading and trailing character or characters. You can specify the character or characters as the following:

- A single char
- A char* array
- An IString (see page 377) object
- An IStringTest (see page 417) object

The default is white space.

stripBlanks

```
static IString stripBlanks(const IString &aString);
```

Strips both leading and trailing white space.

Note: This function is the static version of IString::strip (see page 397), which has been renamed to avoid a duplicate definition.

stripLeading

```
IString &stripLeading();  
IString &stripLeading(char aCharacter);  
IString &stripLeading(const IString &aString);  
IString &stripLeading(const char *pString);  
IString &stripLeading(const IStringTest &aTest);  
static IString stripLeading(const IString &aString, char aChar);  
static IString stripLeading(const IString &aString, const IString &aStringOfChars);  
static IString stripLeading(const IString &aString, const char *pStringOfChars);  
static IString stripLeading(const IString &aString, const IStringTest &aTest);
```

Strips the leading character or characters.

stripLeadingBlanks

```
static IString stripLeadingBlanks(const IString &aString);
```

Strips the leading character or characters.

Note: This function is the static version of IString::stripLeading (see page 398), which has been renamed to avoid a duplicate definition.

stripTrailing

```
static IString stripTrailing(const IString &aString, const char *pStringOfChars);  
IString &stripTrailing();  
IString &stripTrailing(char aCharacter);  
IString &stripTrailing(const IString &aString);  
IString &stripTrailing(const char *pString);  
IString &stripTrailing(const IStringTest &aTest);  
static IString stripTrailing(const IString &aString, char aChar);  
static IString stripTrailing(const IString &aString, const IString &aStringOfChars);  
static IString stripTrailing(const IString &aString, const IStringTest &aTest);
```

Strips the trailing character or characters.

stripTrailingBlanks

```
static IString stripTrailingBlanks(const IString &aString);
```

Strips the trailing character or characters.

Note: This function is the static version of `IString::stripTrailing` (see page 398), which has been renamed to avoid a duplicate definition.

subString

```
IString subString(unsigned startPos) const;
```

```
IString subString(unsigned startPos, unsigned length, char padCharacter = ' ') const;
```

Returns a specified 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.

translate

```
IString &translate(const char *pInputChars, const char *pOutputChars,  
                  char padCharacter = ' ');
```

```
IString &translate(const IString &inputChars, const IString &outputChars,  
                  char padCharacter = ' ');
```

```
IString &translate(const IString &inputChars, const char *pOutputChars,  
                  char padCharacter = ' ');
```

```
IString &translate(const char *pInputChars, const IString &outputChars,  
                  char padCharacter = ' ');
```

```
static IString translate(const IString &aString, const IString &inputChars,  
                        const IString &outputChars, char padCharacter = ' ');
```

```
static IString translate(const IString &aString, const IString &inputChars,  
                        const char *pOutputChars, char padCharacter = ' ');
```

IString

```
static IString translate(const IString &aString, const char *pInputChars,  
                        const IString &outputChars, char padCharacter = ' ');  
  
static IString translate(const IString &aString, const char *pInputChars,  
                        const char *pOutputChars, char padCharacter = ' ');
```

```
protected:  
IString &translate(const char *pInputChars, unsigned inputLen,  
                const char *pOutputChars, unsigned outputLen,  
                char padCharacter);
```

Converts all of the receiver's characters that are in the first specified string to the corresponding character in the second specified string.

upperCase

```
IString &upperCase();  
static IString upperCase(const IString &aString);
```

Translates all lowercase letters in the receiver to uppercase.

word

```
IString word(unsigned wordNumber) const;
```

Returns a copy of the specified white-space-delimited word in the receiver.

wordIndexOfPhrase

```
unsigned wordIndexOfPhrase(const IString &aPhrase, unsigned startWord = 1) const;
```

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.

words

```
IString words(unsigned firstWord, unsigned numWords) const;  
IString words(unsigned firstWord) const;
```

Returns a substring of the receiver that starts at a specified word and consists of a specified number of words. The word separators are copied to the result intact.

x2b

```
IString &x2b();  
static IString x2b(const IString &aString);
```

Converts a string of hexadecimal digits to a string of binary digits. For example, this function changes "a1c" to 101000011100 and "f3" to 11110011.

x2c

```
static IString x2c(const IString &aString);  
IString &x2c();
```

Converts a string of hexadecimal digits to a normal string of characters. For example, this function changes 8 to \x08 and F1F9F9F5 to 1995.

Note: This function is not locale-sensitive; mappings are based on code page IBM-1047 only.

x2d

```
static IString x2d(const IString &aString);
IString &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.

Inherited Public Members

Member	Class	Page	Member	Class	Page
asDebugInfo	IBase	311	operator<<	IBase	312
asString	IBase	311			

Protected Members

applyBitOp

```
IString &applyBitOp(const char *pArg, unsigned argLen, BitOperator op);
```

Implements the bitwise operators &, |, and ^.

buffer

```
IBuffer *buffer() const;
```

Returns the address of the IBuffer (see page 315) referred to by this IString.

change

```
IString &change(const char *pPattern, unsigned patternLen,
               const char *pReplacement, unsigned replacementLen,
               unsigned startPos, unsigned numChanges);
public:
IString &change(const IString &inputString, const IString &outputString,
               unsigned startPos = 1, unsigned numChanges = ( unsigned ) UINT_MAX);
public:
IString &change(const IString &inputString, const char *pOutputString,
               unsigned startPos = 1, unsigned numChanges = ( unsigned ) UINT_MAX);
public:
IString &change(const char *pInputString, const IString &outputString,
               unsigned startPos = 1, unsigned numChanges = ( unsigned ) UINT_MAX);
public:
IString &change(const char *pInputString, const char *pOutputString,
               unsigned startPos = 1, unsigned numChanges = ( unsigned ) UINT_MAX);
public:
static IString change(const IString &aString, const IString &inputString,
                     const IString &outputString, unsigned startPos = 1,
                     unsigned numChanges = ( unsigned ) UINT_MAX);
public:
static IString change(const IString &aString, const IString &inputString,
                     const char *pOutputString, unsigned startPos = 1,
                     unsigned numChanges = ( unsigned ) UINT_MAX);
public:
static IString change(const IString &aString, const char *pInputString,
                     const IString &outputString, unsigned startPos = 1,
                     unsigned numChanges = ( unsigned ) UINT_MAX);
public:
static IString change(const IString &aString, const char *pInputString,
                     const char *pOutputString, unsigned startPos = 1,
                     unsigned numChanges = ( unsigned ) UINT_MAX);
```

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.

IString

The parameters are the following:

inputString

The pattern string as a reference to an object of type IString. The library searches for the pattern string within the receiver's data.

pInputString

The pattern string as null-terminated string. The library searches for the pattern string within the receiver's data.

outputString

The replacement string as a reference to an object of type IString. It replaces the occurrences of the pattern string in the receiver's data.

pOutputString

The replacement string as a null-terminated string. It replaces the occurrences of the pattern string in the receiver's data.

startPos

The position to start the search at within the receiver's data. The default is 1.

numChanges

The number of patterns to search for and change. The default is UINT_MAX, which causes changes to all occurrences of the pattern.

data

```
char *data() const;
```

Returns the address of the contents of the IString.

defaultBuffer

```
static char *defaultBuffer();
```

Returns a pointer to the contents of the nullBuffer data member.

findPhrase

```
unsigned findPhrase(const IString &aPhrase, unsigned startWord, IndexType charOrWord) const;
```

Locates a specified string of words for indexOfWord functions.

indexOfWord

```
unsigned indexOfWord(unsigned wordNumber, unsigned startPos, unsigned numWords) const;  
public:  
unsigned indexOfWord(unsigned wordNumber) const;
```

Returns the index of the specified white-space-delimited word in the receiver. If the word is not found, 0 is returned.

initBuffer

```
IString &initBuffer(const void *p1, unsigned len1,  
                  const void *p2 = 0, unsigned len2 = 0,  
                  const void *p3 = 0, unsigned len3 = 0, char padChar = 0);  
IString &initBuffer(long n);  
IString &initBuffer(unsigned long n);  
IString &initBuffer(long long aLongLong);  
IString &initBuffer(unsigned long long anUnsignedLongLong);  
IString &initBuffer(double d);
```

Resets the contents from a specified buffer or buffers.

insert

```
IString &insert(const char *pInsert, unsigned insertLen,
               unsigned startPos, char padCharacter);
public:
IString &insert(const IString &aString, unsigned index = 0,
               char padCharacter = ' ');
public:
IString &insert(const char *pString, unsigned index = 0,
               char padCharacter = ' ');
public:
static IString insert(const IString &aString, const IString &anInsert,
                     unsigned index = 0, char padCharacter = ' ');
public:
static IString insert(const IString &aString, const char *pInsert,
                     unsigned index = 0, char padCharacter = ' ');
```

Inserts the specified string after the specified location.

isAbbrevFor

```
Boolean isAbbrevFor(const char *pFullString, unsigned fullLen,
                   unsigned minLen) const;
```

If the receiver is a valid abbreviation of the specified string, true is returned. A value of false if returned for null IString.

The parameters are the following:

pFullString

The full string for the abbreviation check. It 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.

isLike

```
Boolean isLike(const char *pPattern, unsigned patternLen,
               char zeroOrMore, char anyChar) const;
public:
Boolean isLike(const IString &aPattern, char zeroOrMore = ' * ',
               char anyChar = '?') const;
public:
Boolean isLike(const char *pPattern, char zeroOrMore = ' * ',
               char anyChar = '?') const;
```

If the receiver matches the specified pattern, which can contain wildcard characters, true is returned. A value of false if returned for null IString.

- You can use the first wildcard character to specify that 0 or more arbitrary characters are accepted. The default wildcard character that does this is *, but you can specify another character when calling IString::isLike. For example:

```
IString( "Allison" ).isLike( "Al*ison" ) -> true
```

- You can use the second wildcard character to specify that a single arbitrary character is accepted. The default wildcard character that does this is ?, but you can specify another character when calling IString::isLike. For example:

```
IString( "istring7.cpp" ).isLike( "i*.?pp" ) -> true
IString( "Not a question!" ).isLike( "*?", '*', '-' ) -> false
```

IString

lengthOf

```
static unsigned lengthOf(const char *p);
```

Returns the length of a C character array.

occurrencesOf

```
unsigned occurrencesOf(const char *pSearchString, unsigned searchLen,  
                        unsigned startPos) const;  
public:  
unsigned occurrencesOf(const IString &aString, unsigned startPos = 1) const;  
public:  
unsigned occurrencesOf(const char *pString, unsigned startPos = 1) const;  
public:  
unsigned occurrencesOf(char aCharacter, unsigned startPos = 1) const;  
public:  
unsigned occurrencesOf(const IStringTest &aTest, unsigned startPos = 1) const;
```

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 (see page 383).

overlayWith

```
IString &overlayWith(const char *pOverlay, unsigned overlayLen,  
                   unsigned index, char padCharacter);  
public:  
IString &overlayWith(const IString &aString, unsigned index = 1,  
                   char padCharacter = ' ');  
public:  
IString &overlayWith(const char *pString, unsigned index = 1,  
                   char padCharacter = ' ');  
public:  
static IString overlayWith(const IString &aString, const IString &anOverlay,  
                           unsigned index = 1, char padCharacter = ' ');  
public:  
static IString overlayWith(const IString &aString, const char *pOverlay,  
                           unsigned index = 1, char padCharacter = ' ');
```

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*).

setBuffer

```
IString &setBuffer(IBuffer *ibuff);
```

Sets the private data member to point to a new IBuffer (see page 315) object.

strip

```
IString &strip(const IStringTest &aTest, IStringEnum::StripMode mode);  
  
IString &strip(const char *p, unsigned len, IStringEnum::StripMode mode);  
public:  
IString &strip();  
public:  
IString &strip(char aCharacter);  
public:  
IString &strip(const IString &aString);  
public:  
IString &strip(const char *pString);  
public:  
IString &strip(const IStringTest &aTest);  
public:  
static IString strip(const IString &aString, char aChar);  
public:
```

```
static IString strip(const IString &aString, const IString &aStringOfChars);
public:
static IString strip(const IString &aString, const char *pStringOfChars);
public:
static IString strip(const IString &aString, const IStringTest &aTest);
```

Strips both leading and trailing character or characters. You can specify the character or characters as the following:

- A single char
- A char* array
- An IString (see page 377) object
- An IStringTest (see page 417) object

The default is white space.

translate

```
IString &translate(const char *pInputChars, unsigned inputLen,
                 const char *pOutputChars, unsigned outputLen,
                 char padCharacter);
public:
IString &translate(const IString &inputChars, const IString &outputChars,
                 char padCharacter = ' ');
public:
IString &translate(const IString &inputChars, const char *pOutputChars,
                 char padCharacter = ' ');
public:
IString &translate(const char *pInputChars, const IString &outputChars,
                 char padCharacter = ' ');
public:
IString &translate(const char *pInputChars, const char *pOutputChars,
                 char padCharacter = ' ');
public:
static IString translate(const IString &aString, const IString &inputChars,
                        const IString &outputChars, char padCharacter = ' ');
public:
static IString translate(const IString &aString, const IString &inputChars,
                        const char *pOutputChars, char padCharacter = ' ');
public:
static IString translate(const IString &aString, const char *pInputChars,
                        const IString &outputChars, char padCharacter = ' ');
public:
static IString translate(const IString &aString, const char *pInputChars,
                        const char *pOutputChars, char padCharacter = ' ');
```

Converts all of the receiver's characters that are in the first specified string to the corresponding character in the second specified string.

I0String

Derivation

```
IBase
 IString
  I0String
```

Header File

i0string.hpp

Objects of the I0String class are functionally equivalent to objects of the class IString (see page 377) 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.

Constructors

```
public:
I0String();
I0String(const IString &aString);
I0String(int);
I0String(unsigned);
I0String(long);
I0String(unsigned long);
I0String(short);
I0String(unsigned short);
IString(long long);
IString(unsigned long long);
I0String(double);
I0String(char);
I0String(unsigned char);
I0String(signed char);
I0String(const char *);
I0String(const unsigned char *);
I0String(const signed char *);
I0String(const void *pBuffer1, unsigned lenBuffer1, char padCharacter = ' ');
I0String(const void *pBuffer1, unsigned lenBuffer1,
    const void *pBuffer2, unsigned lenBuffer2, char padCharacter = ' ');
I0String(const void *pBuffer1, unsigned lenBuffer1,
    const void *pBuffer2, unsigned lenBuffer2,
    const void *pBuffer3, unsigned lenBuffer3, char padCharacter = ' ');
```

You can construct objects of this class in the following ways:

- Construct a null string.
- Construct a string with the ASCII representation of a given numeric value, supporting all flavors of integer and double.
- Construct a string with a copy of the specified character data, supporting ASCIIZ strings, characters, and IStrings. The character data passed is converted to its ASCII representation.
- Construct a string with contents that consist of copies of up to three buffers of arbitrary data (void*). Optionally, you only need to provide the length, in which case, the IString contents are initialized to a specified pad character. The default character is a blank.

These constructors can throw exceptions under the following conditions:

- Memory allocation errors
Many factors dynamically allocate space and these allocation requests may fail. If so, the Application Support 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`.

Public Members

change

```
I0String &change(const char *pPattern, const char *pReplacement,
               unsigned startPos = 0, unsigned numChanges = ( unsigned ) UINT_MAX);

I0String &change(const IString &aPattern, const IString &aReplacement,
               unsigned startPos = 0, unsigned numChanges = ( unsigned ) UINT_MAX);

I0String &change(const IString &aPattern, const char *pReplacement,
               unsigned startPos = 0, unsigned numChanges = ( unsigned ) UINT_MAX);

I0String &change(const char *pPattern, const IString &aReplacement,
               unsigned startPos = 0, unsigned numChanges = ( unsigned ) UINT_MAX);

static I0String change(const IString &aString, const IString &inputString,
                    const IString &outputString, unsigned startPos = 0,
                    unsigned numChanges = ( unsigned ) UINT_MAX);

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

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

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

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 0.

numChanges

The number of patterns to search for and change. The default is 0, which causes changes to all occurrences of the pattern.

IString

charType

```
IStringEnum::CharType charType(unsigned index) const;
```

Returns the type of the character at the specified index.

indexOf

```
unsigned indexOf(const char *pString, unsigned startPos = 0) const;  
unsigned indexOf(const IString &aString, unsigned startPos = 0) const;  
unsigned indexOf(char aCharacter, unsigned startPos = 0) const;  
unsigned indexOf(const IStringTest &aTest, unsigned startPos = 0) const;
```

Returns the byte index of the first occurrence of the specified string within the receiver. If there are no occurrences, -1 is returned. In addition to IStrings, you can also specify a single character or an IStringTest (see page 417).

indexOfAnyBut

```
unsigned indexOfAnyBut(char validChar, unsigned startPos = 0) const;  
unsigned indexOfAnyBut(const IString &aString, unsigned startPos = 0) const;  
unsigned indexOfAnyBut(const char *pValidChars, unsigned startPos = 0) const;  
unsigned indexOfAnyBut(const IStringTest &aTest, unsigned startPos = 0) const;
```

Returns the index of the first character of the receiver that is not in the specified set of characters. If there are no characters, -1 is returned. Alternatively, this function returns the index of the first character that fails the test prescribed by a specified IStringTest (see page 417) object.

indexOfAnyOf

```
unsigned indexOfAnyOf(const char *pSearchChars, unsigned startPos = 0) const;  
unsigned indexOfAnyOf(const IString &searchChars, unsigned startPos = 0) const;  
unsigned indexOfAnyOf(char searchChar, unsigned startPos = 0) const;  
unsigned indexOfAnyOf(const IStringTest &aTest, unsigned startPos = 0) const;
```

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, -1 is returned. Alternatively, this function returns the index of the first character that passes the test prescribed by a specified IStringTest (see page 417) object.

indexOfPhrase

```
unsigned indexOfPhrase(const IString &wordString, unsigned startWord = 1) const;
```

Returns the position of the first occurrence of the specified phrase in the receiver. If the phrase is not found, -1 is returned.

indexOfWord

```
unsigned indexOfWord(unsigned wordNumber) const;
```

Returns the index of the specified white-space-delimited word in the receiver. If the word is not found, -1 is returned.

insert

```
static IString insert(const IString &aString, const IString &anInsert,  
                     unsigned index = ( unsigned ) UINT_MAX, char padCharacter = ' ');  
  
IString &insert(const IString &aString, unsigned index = ( unsigned ) UINT_MAX,  
              char padCharacter = ' ');  
  
IString &insert(const char *pString, unsigned index = ( unsigned ) UINT_MAX,  
              char padCharacter = ' ');  
  
static IString insert(const IString &aString, const char *pInsert,
```

```
unsigned index = ( unsigned ) UINT_MAX, char padCharacter = ' ');
```

Inserts the specified string at the specified location.

lastIndexOf

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

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

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

unsigned lastIndexOf(const IStringTest &aTest,
    unsigned startPos = ( unsigned ) ( UINT_MAX-1 )) const;
```

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 $0 \leq x \leq \text{startPos}$ or `IOString::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, -1 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, the search target must occur at the beginning of the string.

lastIndexOfAnyBut

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

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

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

unsigned lastIndexOfAnyBut(const IStringTest &aTest,
    unsigned endPos = ( unsigned ) ( UINT_MAX-1 )) const;
```

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, -1 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, the search target must occur at the beginning of the string.

lastIndexOfAnyOf

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

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

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

unsigned lastIndexOfAnyOf(
    const IStringTest &aTest,
    unsigned endPos = ( unsigned ) ( UINT_MAX-1 )) const;
```

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, -1 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, the search target must occur at the beginning of the string.

occurrencesOf

```
unsigned occurrencesOf(const IString &aString, unsigned startPos = 0) const;
unsigned occurrencesOf(const char *pString, unsigned startPos = 0) const;
unsigned occurrencesOf(char aCharacter, unsigned startPos = 0) const;
unsigned occurrencesOf(const IStringTest &aTest, unsigned startPos = 0) const;
```

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 (see page 383).

operator[]

```
const char &operator[](unsigned index) const;
char &operator[](unsigned index);
```

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.

overlayWith

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

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

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

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

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*).

remove

```
I0String &remove(unsigned startPos);

I0String &remove(unsigned startPos, unsigned numChars);

static I0String remove(const IString &aString, unsigned startPos);

static I0String remove(const IString &aString, unsigned startPos,
                    unsigned numChars);
```

Deletes the specified portion of a string 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.

subString

```
IString subString(unsigned startPos) const;
IString subString(unsigned startPos, unsigned len, char padCharacter = ' ') const;
```

Returns a specified portion of a string (that is, 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.

Note: The following table shows the IString public members inherited from IBase. Members inherited from IString are not shown. See "IString" on page 377 for information on the members of IString.

Inherited Public Members

Member	Class	Page	Member	Class	Page
asDebugInfo	IBase	311	operator<<	IBase	312
asString	IBase	311			

Protected Members

adjustArg

```
static unsigned adjustArg(unsigned index);
```

Adjusts the specified index from 0- to 1-based.

adjustResult

```
static unsigned adjustResult(unsigned index);
```

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

Note: IString inherits protected members from IBase. See "IBase" on page 311 for information on the members of IBase.

IStringEnum

Derivation

IStringEnum

Header File

istrenum.hpp

The IStringEnum class serves as a repository for enumeration types related to the IString class. The Application Support Class Library places these enumeration types here so they can easily be shared between code that implements the classes IString (see page 377), IBuffer (see page 315), and IDBCSBuffer (see page 325).

Enumerations

CharType

```
public:
typedef enum
{
    sbcs,
    dbcs1 = 1, mbcs1 = 1,
    dbcs2 = 2, mbcs2 = 2,
    mbcs3 = 3,
    mbcs4 = 4
} CharType;
```

This typedef specifies the various types of characters that comprise an IString:

sbcs Contains single-byte character set (SBCS) characters.

dbcs1 Contains the first byte of a double-byte character support (DBCS) character.

dbcs2 Contains the second byte of a double-byte character support (DBCS) character.

mbcs1 Contains the first byte of a multi-byte character support (MBCS) character.

mbcs2 Contains the second byte of a multi-byte character support (MBCS) character.

mbcs3 Contains the third byte of a multi-byte character support (MBCS) character.

mbcs4 Contains the fourth byte of a multi-byte character support (MBCS) character.

Note: Code for MBCS must be the same as the length of the corresponding characters.

StripMode

```
public:
typedef enum {leading, trailing, both} StripMode;
```

This typedef defines the mode of various functions that strip characters from an IString:

leading

Strips the leading characters.

trailing

Strips the trailing characters.

both

Strips both the leading and trailing characters.

IStringParser

Derivation

```
IBase
  IVBase
    IStringParser
```

Header File

```
istparse.hpp
```

Objects of this class enable you to parse the content of an IString (see page 377) and place portions of the string into other strings. You can limit the parsing of a string by specifying the following:

- Patterns that must be matched
- Relative or absolute column numbers

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: <div style="margin-left: 20px;"> <p>const char*</p> <p>Searches for the sequence of characters described by the character array.</p> <p>const IString</p> <p>Searches for the sequence of characters described by the string. Note that this differs from the treatment of a non-const IString, as described at the start of this list.</p> </div> |

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::SkipWords

An object of this class skips a given number of tokens.

You can also use the left-shift operator with an unsigned numeric parameter. It 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.

Nested Classes

IStringParser::SkipWords

Constructors

```
protected:  
IStringParser(const IString &text);
```

```
protected:  
IStringParser(const IStringParser &parser);
```

You can construct objects of this class by providing either of the following:

- An `IString` object
- Another `IStringParser` object

Use these constructors to specify the text string to parse. Usually, you construct `IStringParser` objects by applying the right-shift operator to a string. These constructors are protected to prevent you from creating objects, except via the right-shift operators. Such prevention is necessary because of the nature of string parser objects. Because they hold references to operands, it is unwise to let the objects persist beyond the scope of those operands.

Public Members

operator<<

```
IStringParser &operator<<(unsigned position);
```

Changes the parser text position to an absolute column number. This is a left-shift operator.

operator>>

```
IStringParser &operator>>(int delta);
IStringParser &operator>>(IString &token);
IStringParser &operator>>(const IString &pattern);
IStringParser &operator>>(const char *pattern);
IStringParser &operator>>(char pattern);
IStringParser &operator>>(const IStringTest &test);
IStringParser &operator>>(unsigned delta);
IStringParser &operator>>(Command cmd);
IStringParser &operator>>(const SkipWords &skipObj);
```

Parses the text string. The right-shift operator is the primary function for parsing the text string. The Application Support 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.

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."
```

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.

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"
```

cmd 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 (see page 416) to specify the parsing token.

skipObj Skips the next n tokens in the parser text, where n is the number of words specified when constructing the IStringParser::SkipWords (see page 416) object.

IStringParser::SkipWords

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.

Inherited Public Members

Member	Class	Page	Member	Class	Page
asDebugInfo	IBase	311	asString	IVBase	314
asDebugInfo	IVBase	314	operator<<	IBase	312
asString	IBase	311	operator<<	IVBase	314

Enumerations

Command

```
public:  
enum Command { reset, skip };
```

These enumerators specify special purpose parsing tokens:

reset Resets the parser position to 1.
skip Causes the parser to skip one token (that is, a word) in the input text.

IStringParser::SkipWords

Derivation

```
IBase  
IVBase  
IStringParser::SkipWords
```

Header File

```
istparse.hpp
```

Objects of the nested class IStringParser::SkipWords skip a specified number of tokens (that is, words or phrases) in the input text without assigning those tokens to output strings. Use these objects when parsing text with the class IStringParser (see page 413).

Constructors

```
SkipWords(unsigned long numWords = 1);
```

You can construct objects of this class by specifying the number of tokens to skip. The default is one token.

Public Members

numberOfWords

```
unsigned numberOfWords() const;
```

Returns the number of tokens to skip.

Inherited Public Members

Member	Class	Page	Member	Class	Page
asDebugInfo	IBase	311	asString	IVBase	314
asDebugInfo	IVBase	314	operator<<	IBase	312
asString	IBase	311	operator<<	IVBase	314

IStringTest

Derivation

```
IBase
IVBase
IStringTest
```

Header File

```
istrtest.hpp
```

The IStringTest class defines the basic protocol for test objects that you can pass to IStrings (see page 377) or IOStrings (see page 405) 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 Application Support Class Library provides a derived template class, IStringTestMemberFn (see page 418), 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 (see page 418) 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.

Constructors

```
public:
IStringTest(CFunction &cFunc);
IStringTest(CPPFunction &cppFunc);

protected:
IStringTest(FnType type, void *userData);
```

You can construct an object of this class with a pointer to the C function to be used to implement the member function IStringTest::test (see page 418).

This class also provides a protected constructor, which derived classes can use to reuse the space for the C/C++ function pointer.

Public Members

test

```
virtual Boolean test(int c) const;
```

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.

Inherited Public Members

Member	Class	Page	Member	Class	Page
asDebugInfo	IBase	311	asString	IVBase	314
asDebugInfo	IVBase	314	operator<<	IBase	312
asString	IBase	311	operator<<	IVBase	314

Enumerations

FnType

```
public:  
enum 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** C++ **const** member function

IStringTestMemberFn

Derivation

- IBase
- IVBase
- IStringTest
- IStringTestMemberFn

Header File

```
istrtest.hpp
```

The Application Support Class Library provides the template class IStringTestMemberFn as an IStringTest-type wrapper for particular C++ member functions. You can use such member functions in conjunction with functions from IString (see page 377) and I0String (see page 405) that accept an IStringTest (see page 417) 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.

Constructors

```
IStringTestMemberFn(const T &object, ConstFn constFn);
IStringTestMemberFn(T &object, NonconstFn nonconstFn);
```

You can construct objects of this class by using the following:

- The constructor supporting **const** member functions.
- The constructor supporting non-**const** member functions. You must specify a non-**const** member 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 library applies this member function to the specified object to test each character passed to the test function of this class. The member function must accept a single integer parameter and return Boolean.

Public Members

test

```
virtual Boolean test(int c) const;
```

Overridden to dispatch a member function against an object.

Inherited Public Members

Member	Class	Page	Member	Class	Page
asDebugInfo	IBase	311	operator<<	IBase	312
asDebugInfo	IVBase	314	operator<<	IVBase	314
asString	IBase	311	test	IStringTest	418
asString	IVBase	314			

Chapter 61. IApplication

Derivation

```

IBase
IVBase
IApplication

```

Inherited By

```

ICurrentApplication

```

Header File

```

iapp.hpp

```

Members

Member	Page
Constructor	424
adjustPriority	422
asDebugInfo	422
asString	422
current	423
currentPID	424
id	424
setId	425
setPriority	423
~IApplication	424

The IApplication class represents processes. The Application Support Class Library only supports the currently executing application, the single object of the derived class ICurrentApplication (p. 443). IApplication::current provides access to that object.

This class maintains a static pointer to the C++ object representing the currently executing application. It does so to implement the static member function current (p. 423), which returns a reference to the ICurrentApplication object. The set of functions that you can apply to the current application is different from the set of functions you can apply to other applications. ICurrentApplication defines these functions.

Public Functions

Diagnostics

Use these members for diagnostic purposes. They return an IString representation of an object of this class.

IApplication

asDebugInfo

Returns a representation of the application as debug information. The representation is returned as an IString with the following contents:

```
IApplication(IVBase(@addr),id=pid)
```

This represents the following:

addr The address of the object (in hexadecimal).

pid The process ID (in hexadecimal).

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

asString

Returns the string "IApplication(pid)", where *pid* represents the process identifier.

```
virtual IString  
    asString() const;
```

Priority

Use these members to control the priority of the application (and its threads).

Note: While you can set priorities using this class, only threads actually have a priority. As a result, you need to use functions of the IThread (p. 535) class to query the priority of an application's threads.

adjustPriority

Adjusts the priority level of all of the application's threads by some amount. An optional flag specifies whether the library also modifies the priority of descendent processes.

```
virtual IApplication&  
    adjustPriority( long adjustment,  
                  Boolean setDescendents = false);
```

adjustment

Long value that represents the adjustment delta. This value must be in the range of -31 to 31.

setDescendents

Boolean that determines whether the library also modifies the priority of descendent processes.

OS/390: This member has no effect.

Windows (Win32s): Win32s does not support multiple threads or thread priorities. Thus, IThread::adjustPriority and IApplication::adjustPriority have no effect.

Motif: This member function calls IThread::adjustPriority for the current (and only) thread. The AIX release of the Application Support Class Library does not support multiple threads or thread priorities; thus, IThread::adjustPriority and IApplication::adjustPriority have no effect.

Exceptions	
IAccessError	The priority was not adjusted. The adjustment delta must be in the range of -31 to 31.

setPriority

Sets the priority (p. 425) class and level of all of the application's threads to the specified value. An optional flag specifies whether the library also modifies the priority of descendent processes.

```
virtual IApplication&
    setPriority( PriorityClass priorityClass,
               long priorityLevel = 0,
               Boolean setDescendents = false);
```

priorityClass

Enumeration that identifies the priority class to set.

priorityLevel

Long value that represents the new priority level of all the application's threads for the priority class. This value must be in the range of -31 to 31.

setDescendents

Boolean that determines whether the library also modifies the priority of descendent processes.

OS/390: This member has no effect.

Windows (Win32s): Win32s does not support multiple threads or thread priorities. Thus, IThread::setPriority and IApplication::setPriority have no effect.

Motif: This member function calls IThread::setPriority for the current (and only) thread. The AIX release of the Application Support Class Library does not support multiple threads or thread priorities; thus, IThread::setPriority and IApplication::setPriority have no effect.

Exceptions	
IAccessError	The priority was not set. The priority level may be invalid.

Process Information

Use these members to get additional information about a process, such as the process identifier, or to access the object for the current process.

current

Returns a reference to the current application, which is an object of the class ICurrentApplication (p. 443).

```
static ICurrentApplication&
    current();
```

IApplication

currentPID

Returns the current process identifier value.

```
static IProcessId  
currentPID();
```

id

Returns this object's process identifier value.

```
virtual IProcessId  
id() const;
```

Inherited Public Functions

IVBase		
asDebugInfo	asString	

IBase		
asDebugInfo	messageFile	setMessageFile
asString	messageText	version

Protected Functions

Constructors

The constructors and destructor for this class are protected. You must derive from this class if you want to use it to represent an application.

IApplication

You can only construct objects of this class with the process identifier for the process that the object will represent.

```
IApplication( const IProcessId& id);
```

id Reference to the process identifier for the process that the object will represent.

~IApplication

```
virtual  
~IApplication();
```

Setting Process Information

Use these members to set the information on a process, such as the process identifier of this object.

setId

Sets this object's process identifier. You call this function in your derived class when you start a process in order to save the process identifier for this object.

```
virtual IApplication&
    setId( const IProcessId& id);
```

id Reference to the process identifier to save.

Inherited Protected Data

IBase		
recoverable	unrecoverable	

PriorityClass

```
PriorityClass {
    noChange,          idleTime,          regular,          timeCritical,
    foregroundServer
};
```

These enumerators specify one of the priority classes, such as when using `setPriority`: (p. 423)

noChange

Specifies no change to the thread priority.

idleTime

Specifies the lowest priority. This priority only gets processing time when there is no other work to do.

regular

Specifies the default priority. Most threads belong to this class.

timeCritical

Specifies the highest priority. Use this priority when response time is critical.

foregroundServer

Specifies that a program running on a server takes precedence over other processes on the server.

Presentation Manager: For additional information, see the *OS/2 3.0 Control Program Guide and Reference*.

Windows (Win32s): The library provides this enumeration for portability purposes only. Win32s does not support priority adjustment.

Motif: The library provides this enumeration for portability purposes only. The AIX release of the Application Support Class Library does not support priority adjustment.

Chapter 62. Decimal Classes

IBinaryCodedDecimal

Derivation

Inherits from none.

Inherited By

None.

Header File

idecimal.hpp

Members

Objects of the IBinaryCodedDecimal class represent numerical quantities accurately, especially in business and commercial applications for financial calculations.

This class provides functions to obtain information about, compare, and manipulate IBinaryCoded Decimal objects.

Constructors

IBinaryCodedDecimal

The IBinaryCodedDecimal object can be constructed in many different ways. If the number of digits and precisions are provided, the constructed object has the provided digit and precision; otherwise default values are assumed according to the type of the value. If a value is provided during construction, the constructed object has the given value with the appropriate digit and precision; otherwise the constructed object will have a value zero. An exception is thrown if the number of digits in the integral part is larger than the supported value (for example, DEC_DIG) or the provided digit and/or precision are out of range. The valid values should satisfy the following:

```

DIGITS      - the given number of digits
PRECISIONS  - the given number of precisions

0    <    DIGITS          <=    DEC_DIG
0    <=   PRECISIONS      <=    DEC_DIG
      PRECISIONS          <=    DIGITS

```

All the macro constants are defined in idecimal.hpp.

```
IBinaryCodedDecimal( long aLong);
```

IBinaryCodedDecimal

Constructs the IBinaryCodedDecimal object to have a value *aLong* with DFT_LNG_DIG number of digits and zero number of precisions.

```
IBinaryCodedDecimal();
```

This is a default constructor. The constructed object has a value zero with attributes of DFT_DIG number of digits and DFT_PREC number of precisions.

```
IBinaryCodedDecimal( const char* val);
```

Constructs the IBinaryCodedDecimal object according to the value represented in the provided string. The number of digits and precisions are set accordingly. The leading and trailing zeros are used when setting up the digit and precision attributes, for example, "00123.4560" has 9 digits and 4 precisions.

```
IBinaryCodedDecimal( int anInt);
```

Constructs the IBinaryCodedDecimal object to have a value *anInt* with DFT_INT_DIG number of digits and zero number of precisions.

```
IBinaryCodedDecimal( unsigned int auInt);
```

Constructs the IBinaryCodedDecimal object to have a value *auInt* with DFT_INT_DIG number of digits and zero number of precisions.

```
IBinaryCodedDecimal( unsigned long auLong);
```

Constructs the IBinaryCodedDecimal object to have a value *auLong* with DFT_LNG_DIG number of digits and zero number of precisions.

```
IBinaryCodedDecimal( long long aLongLong);
```

Constructs the IBinaryCodedDecimal object to have a value *aLongLong* with DFT_LNG_DIG number of digits and zero number of precisions.

```
IBinaryCodedDecimal( unsigned long long auLongLong);
```

Constructs the IBinaryCodedDecimal object to have a value *auLongLong* with DFT_LNG_DIG number of digits and zero number of precisions.

```
IBinaryCodedDecimal( float afloat);
```

Constructs the IBinaryCodedDecimal object to have a value *afloat*.

Note: Precision loss could occur during conversion.

```
IBinaryCodedDecimal( double aDouble);
```

Constructs the IBinaryCodedDecimal object to have a value *aDouble*.

Note: Precision loss could occur during conversion.

```
IBinaryCodedDecimal( long double aDouble);
```

Constructs the IBinaryCodedDecimal object to have a value *aDouble*.

Note: Precision loss could occur during conversion.

```
IBinaryCodedDecimal( unsigned int nDig,
                    unsigned int nPrec);
```

Constructs the IBinaryCodedDecimal object to have a value zero with *nDig* number of digits and *nPrec* number of precisions.

```
IBinaryCodedDecimal( unsigned int nDig,
                    unsigned int nPrec,
                    int anInt);
```

Constructs the IBinaryCodedDecimal object to have a value *anInt* with *nDig* number of digits and *nPrec* number of precisions.

```
IBinaryCodedDecimal( unsigned int nDig,
                    unsigned int nPrec,
                    unsigned int auInt);
```

Constructs the IBinaryCodedDecimal object to have a value *auInt* with *nDig* number of digits and *nPrec* number of precisions.

```
IBinaryCodedDecimal( unsigned int nDig,
                    unsigned int nPrec,
                    long aLong);
```

Constructs the IBinaryCodedDecimal object to have a value *aLong* with *nDig* number of digits and *nPrec* number of precisions.

```
IBinaryCodedDecimal( unsigned int nDig,
                    unsigned int nPrec,
                    unsigned long auLong);
```

Constructs the IBinaryCodedDecimal object to have a value *auLong* with *nDig* number of digits and *nPrec* number of precisions.

```
IBinaryCodedDecimal( unsigned int nDig,
                    unsigned int nPrec,
                    long long aLongLong);
```

Constructs the IBinaryCodedDecimal object to have a value *aLongLong* with *nDig* number of digits and *nPrec* number of precisions.

```
IBinaryCodedDecimal( unsigned int nDig,  
                     unsigned int nPrec,  
                     unsigned long long auLongLong);
```

Constructs the IBinaryCodedDecimal object to have a value *auLongLong* with *nDig* number of digits and *nPrec* number of precisions.

```
IBinaryCodedDecimal( unsigned int nDig,  
                     unsigned int nPrec,  
                     float afloat);
```

Constructs the IBinaryCodedDecimal object to have a value *afloat* with *nDig* number of digits and *nPrec* number of precisions.

Note: Precision loss could occur during conversion.

```
IBinaryCodedDecimal( unsigned int nDig,  
                     unsigned int nPrec,  
                     double aDouble);
```

Constructs the IBinaryCodedDecimal object to have a value *aDouble* with *nDig* number of digits and *nPrec* number of precisions.

Note: Precision loss could occur during conversion.

```
IBinaryCodedDecimal( unsigned int nDig,  
                     unsigned int nPrec,  
                     long double aDouble);
```

Constructs the IBinaryCodedDecimal object to have a value *aDouble* with *nDig* number of digits and *nPrec* number of precisions.

Note: Precision loss could occur during conversion.

Public Members

Comparisons

operator !=

This operator is used to compare two IBinaryCodedDecimal objects. Returns 1 if the represented values in both objects are not the same; otherwise returns 0.

operator <

This operator is used to compare two IBinaryCodedDecimal objects. Returns 1 if the left operand represents a smaller value than the right operand; otherwise returns 0.

operator <=

This operator is used to compare two IBinaryCodedDecimal objects. Returns 1 if the left operand represents a smaller or same value than the right operand; otherwise returns 0.

operator ==

This operator is used to compare two IBinaryCodedDecimal objects. Returns 1 if the represented values in both object are the same; otherwise returns 0.

operator >

This operator is used to compare two IBinaryCodedDecimal objects. Returns 1 if the left operand represents a larger value than the right operand; otherwise returns 0.

operator >=

This operator is used to compare two IBinaryCodedDecimal objects. Returns 1 if the left operand represents a larger or same value than the right operand; otherwise returns 0.

Manipulations**operator !**

Returns 1 if the IBinaryCodedDecimal object contains the value zero; otherwise returns 0.

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

operator *=

This operator is used to perform compound assignment between two IBinaryCodedDecimal objects. The right IBinaryCodedDecimal object is converted to the utility class, IDecimalUtil first. The left operand is multiplied by the right operand, the result is stored back to the left operand.

Note: IBinaryCodedDecimal uses the utility class IDecimalUtil to perform its operations.

```
void
    operator *=( const IDecimalUtil& dut);
```

operator +

This is a unary plus operator. The copy of the original object is returned.

```
IBinaryCodedDecimal
    operator +() const;
```

operator ++

This is an increment operator.

```
IBinaryCodedDecimal
    operator ++();
```

This is a prefix increment operator. The value being represented is incremented by 1 and the object with the new value is returned.

```
IBinaryCodedDecimal  
    operator ++( int);
```

This is a postfix increment operator. The original object is returned. As a side effect, the value being represented is incremented by 1.

operator +=

This operator is used to perform compound assignment between two IBinaryCodedDecimal objects. The right IBinaryCodedDecimal object is converted to the utility class, IDecimalUtil first. The right operand is added to the left operand, the result is stored back to the left operand.

Note: IBinaryCodedDecimal uses the utility class IDecimalUtil to perform its operations.

```
void  
    operator +=( const IDecimalUtil& dut);
```

operator -

This is a unary minus operator. The negation of the object is returned.

```
IBinaryCodedDecimal  
    operator -() const;
```

operator --

This is a decrement operator.

```
IBinaryCodedDecimal  
    operator --();
```

This is a prefix decrement operator. The value being represented is decremented by 1 and the object with the new value is returned.

```
IBinaryCodedDecimal  
    operator --( int);
```

This is a postfix decrement operator. The original object is returned. As a side effect, the value being represented is decremented by 1.

operator -=

This operator is used to perform compound assignment between two IBinaryCodedDecimal objects. The right IBinaryCodedDecimal object is converted to the utility class, IDecimalUtil first. The left operand is subtracted from the right operand, the result is stored back to the left operand.

Note: IBinaryCodedDecimal uses the utility class IDecimalUtil to perform its operations.

```
void  
    operator -=( const IDecimalUtil& dut);
```

operator /=

This operator is used to perform compound assignment between two IBinaryCodedDecimal objects. The right IBinaryCodedDecimal object is converted to the utility class, IDecimalUtil first. The left operand is divided by the right operand, the result is stored back to the left operand.

Note: IBinaryCodedDecimal uses the utility class IDecimalUtil to perform its operations.

```
void
    operator /=( const IDecimalUtil& dut);
```

operator =

Different assignment operators are supported for assigning different data types to the IBinaryCodedDecimal object.

```
IBinaryCodedDecimal&
    operator =( double aDouble);
```

Assigns the double value to the IBinaryCodedDecimal object.

```
IBinaryCodedDecimal&
    operator =( const IBinaryCodedDecimal& bcd);
```

Assignment between IBinaryCodedDecimal objects.

```
IBinaryCodedDecimal&
    operator =( const IDecimalUtil& dut);
```

Assigns the utility class object, IDecimalUtil, to the IBinaryCodedDecimal object.

```
IBinaryCodedDecimal&
    operator =( int anInt);
```

Assigns the integer value to the IBinaryCodedDecimal object.

```
IBinaryCodedDecimal&
    operator =( unsigned auInt);
```

Assigns the unsigned int type value to the IBinaryCodedDecimal object.

```
IBinaryCodedDecimal&
    operator =( long aLong);
```

Assigns the long type value to the IBinaryCodedDecimal object.

```
IBinaryCodedDecimal&
    operator =( unsigned long auLong);
```

Assigns the unsigned long type value to the IBinaryCodedDecimal object.

```
IBinaryCodedDecimal&
    operator =( long long aLongLong);
```

Assigns the long long type value to the IBinaryCodedDecimal object.

IBinaryCodedDecimal

```
IBinaryCodedDecimal&  
operator =( unsigned long long auLongLong);
```

Assigns the unsigned long long type value to the IBinaryCodedDecimal object.

```
IBinaryCodedDecimal&  
operator =( float aFloat);
```

Assigns the float type value to the IBinaryCodedDecimal object.

```
IBinaryCodedDecimal&  
operator =( long double aDouble);
```

Assigns the long double type value to the IBinaryCodedDecimal object.

Queries

cData

Returns a pointer that points to the internal buffer which contains the binary coded decimal value in packed decimal format, for example two digits per byte, the last right nibble contains the sign.

This expression `(digitsOf() >> 1) + 1` identifies the number of bytes to store the binary coded decimal value.

Note: The buffer is not null terminated.

```
const char*  
cData() const;
```

digitsOf

Returns the number of digits represented by the binary coded decimal value.

```
int  
digitsOf() const;
```

isNegative

Returns true if the IBinaryCodeDecimal object represents a negative value; otherwise returns false.

```
const bool  
isNegative() const;
```

isPositive

Returns true if the IBinaryCodeDecimal object represents a positive value; otherwise returns false.

Note: Zero is considered as a positive value.

```
const bool  
isPositive() const;
```


precisionOf

Returns the number of precisions.

```
int
    precisionOf() const;
```

Type Conversions**asDouble**

Returns the binary coded decimal as a double value.

```
double
    asDouble() const;
```

asLong

Returns the binary coded decimal as a long type value. Any numbers after the decimal point are truncated. If the value in the binary coded decimal is too large or too small to be represented as a long type, the returned value is undefined.

```
long
    asLong() const;
```

asLongLong

Returns the binary coded decimal as a long long type value. Any numbers after the decimal point are truncated. If the value in the binary coded decimal is too large or too small to be represented as a long type, the returned value is undefined.

```
long long
    asLongLong() const;
```

asString

Returns the binary coded decimal as IString representation.

```
IString
    asString() const;
```

Protected Functions**Streaming****readFromStream**

Classes that inherit from IBinaryCodedDecimal can call readFromStream() to get information from the data stream *fromWhere* for the IBinaryCodedDecimal class.

```
void
    readFromStream( IDataStream& fromWhere);
```

OS/390: This member function is not supported.

Decimal

writeToStream

Classes that inherit from `IBinaryCodedDecimal` can call `writeToStream()` to output the information for the `IBinaryCodedDecimal` class to the data stream *toWhere*.

```
void  
    writeToStream( IDataStream& toWhere) const;
```

OS/390: This member function is not supported.

Decimal

Derivation

Inherits from `decimalBase`.

Inherited By

None.

Header File

`idecimal.hpp`

The decimal class offers faster performance than the `IBinaryCodedDecimal` class and was developed to be compatible with the decimal data type in C, PL/I, and COBOL. Objects of the decimal class represent numerical quantities accurately, especially in business and commercial applications for financial calculations.

This class provides functions to obtain information about, compare, and manipulate decimal objects.

Constructors

Decimal

The decimal object can be constructed in many different ways. The number of digits must be specified. Specifying the precision is optional; the default value for precision is 0. The constructed object will have either the provided number of digits and precision or the provided number of digits and 0 precision.

If a value for the object is provided during construction, the constructed object has the given value with the appropriate digit and precision; otherwise the constructed object will have a value of zero.

The range of digits and precision is:

DIGITS - the given number of digits (integral + any decimal digits excluding sign and decimal points)
 PRECISION - the given precision (i.e. decimal digits)

1 <= DIGITS <= 31
 0 <= PRECISION <= DIGITS

Specifying a value for PRECISION is optional; the default is zero.

Decimal numbers must be in quotation marks.

The number of integral digits cannot be greater than

DIGITS - PRECISION

otherwise an exception is thrown.

If the number of decimal digits is greater than PRECISION, extra digits to the right of the decimal parts will be dropped with no rounding.

If the provided digits and/or precision are out of range, the maximum values are used and no exception is thrown. For example:

`decimal<36,34>op1==>decimal<31,31>op1`

`decimal<5,29>op1====>decimal<5,5>op1`

`decimal<36,5>op1====>decimal<31,5>op1`

`decimal<nDig,nPrec>::decimal(void);`

Constructs the decimal object to have a value of zero with *nDig* number of digits and *nPrec* digits of precisions.

`decimal<nDig,nPrec>::decimal(char*);`

Constructs the decimal object according to the value represented in the provided string. The number of digits and precision are set accordingly. The leading and trailing zeros are used when setting up the digit and precision attributes, for example, "00123.4560" has 9 digits including 4 digits of precision.

`decimal<nDig,nPrec>::decimal(int);`

Constructs the decimal object with *nDig* number of digits and *nPrec* digits of precision by accepting a value of type *int*.

`decimal<nDig,nPrec>::decimal(IBinaryCodedDecimal);`

Constructs the decimal object with *nDig* number of digits and *nPrec* digits of precision by accepting an *IBinaryCodedDecimal* type value.

```
decimal<nDig,nPrec>::decimal(decimal);
```

This is a copy constructor. Constructs the decimal object with *nDig* number of digits and *nPrec* digits of precision by accepting an *decimal* type value.

Public Members

Comparisons

Comparison operators can compare either two decimal objects or a decimal and an integer.

operator !=

Returns 1 if the represented values in both object are not the same; otherwise returns 0.

operator <

Returns 1 if the left operand represents a lesser value than the right operand; otherwise returns 0.

operator <=

Returns 1 if the left operand represents a value lesser than or equal to the right operand; otherwise returns 0.

operator ==

Returns 1 if the represented values in both object are the same; otherwise returns 0.

operator >

Returns 1 if the left operand represents a greater value than the right operand; otherwise returns 0.

operator >=

Returns 1 if the left operand represents a greater or same value than the right operand; otherwise returns 0.

Manipulations

Operands in all arithmetic operations must be between two decimal numbers or between a decimal and an integer.

operator !

Returns 1 if the Decimal object contains the value zero; otherwise returns 0.

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

operator *

This operator yields the product of its operands.

operator *=

This operator is used to perform compound assignment. The left operand is multiplied by the right operand, the result is stored back to the left operand.

operator +

This operator yields the sum of its operands.

operator ++

This is an increment operator.

```
operator ++();
```

This is a prefix increment operator. The value being represented is incremented by 1 and the object with the new value is returned.

```
operator ++(int);
```

This is a postfix increment operator. The original object is returned. As a side effect, the value being represented is incremented by 1.

operator +=

This operator is used to perform compound assignment. The right operand is added to the left operand, the result is stored back to the left operand.

operator -

This operator yields the difference of its operands.

operator --

This is a decrement operator.

```
operator --();
```

This is a prefix decrement operator. The value being represented is decremented by 1 and the object with the new value is returned.

```
operator --(int);
```

This is a postfix decrement operator. The original object is returned. As a side effect, the value being represented is decremented by 1.

operator -=

This operator is used to perform compound assignment. The left operand is subtracted from the right operand, the result is stored back to the left operand.

operator /

This operator yields the quotient of its operands.

Decimal

operator /=

This operator is used to perform compound assignment. The left operand is divided by the right operand, the result is stored back to the left operand.

operator =

This operator can assign different data types to the decimal object:

- an IBinaryCodedDecimal value to the decimal object.
- an integer value to the decimal object.
- an unsigned int type value to the decimal object.
- a string value to the decimal object.

Streaming

operator <<

Output a decimal object with the ostream operator <<. The sign (+ or -) of the decimal is also printed and the number is zero filled to the specified DIGITS and PRECISION. If a sign was not specified when constructing, + sign is the default.

```
decimal<8,2> op1 = "345.3";
```

```
cout << op1 << endl;    // +000345.30 is printed
```

operator >>

Assign a number to a decimal object using the input operator >>.

```
decimal<8,2> op1 = "+345.3";
```

```
cin >> op1;                // when you type 123.45 as input,  
                           // op1 will become 123.45
```

Queries

cData

Returns a pointer that points to the internal buffer which contains the decimal value in packed decimal format, for example two digits per byte, the last right nibble contains the sign.

This expression (digitsOf() >> 1) + 1 identifies the number of bytes to store the decimal value.

Note: The buffer is not null terminated.

```
char*  
    cData() const;
```

digitsOf

Returns the number of digits represented by the decimal value.

```
int  
    digitsOf() const;
```

isNegative

Returns one (1) if the decimal object represents a negative value; otherwise returns zero (0).

```
int
    isNegative() const;
```

isPositive

Returns one (1) if the decimal object represents a positive value; otherwise returns zero (0).

Note: Zero is considered as a positive value.

```
int
    isPositive() const;
```

precisionOf

Returns the number of digits of precision.

```
int
    precisionOf() const;
```

Type Conversions**asBCD**

Returns the decimal object as an `IBinaryCodedDecimal` representation.

```
IBinaryCodedDecimal
    asBCD() const;
```

asString

Returns the decimal object as an `IString` representation.

```
IString
    asString() const;
```

The sign (+ or -) of the decimal is also included, and the number is zero filled to the specified `DIGITS` and `PRECISION`. If a sign was not specified when constructing, + sign is the default.

Chapter 63. ICurrentApplication

Derivation

```
IBase
IVBase
  IApplication
    ICurrentApplication
```

Inherited By

None.

Header File

iapp.hpp

Members

Member	Page
Constructor	445
argc	443
argv	444
asDebugInfo	444
exit	444
pib	446
run	445
setArgs	444
~ICurrentApplication	445

The ICurrentApplication class represents the program that is currently running. You are limited to a single object of this class. To obtain a reference to the object, use the static function IApplication::current (p. 423). The object of this class contains information that the Application Support Class Library maintains for each running application.

Public Functions

Arguments

Use these members to access or set the program arguments that can be passed to a program when it is executed.

argc

Obtains the number of application arguments.

```
virtual int
  argc() const;
```

argv

Obtains the specified argument that is passed to the application.

```
virtual IString  
    argv( int argumentNumber) const;
```

argumentNumber

Integer value that represents the specified argument.

setArgs

Sets the program arguments. Call this member function from your application's main function, passing its *argc* and *argv* argument values.

```
virtual ICurrentApplication&  
    setArgs( int argc,  
            const char * const argv [ ]);
```

argc Integer value that states the number of arguments.

argv Pointer to an array of arguments.

Diagnostics

Use these members for diagnostic purposes. They return an IString representation of an object of this class.

asDebugInfo

Use this function to return a diagnostic representation of the object. The information that is returned is composed of the following:

- Number of arguments
- Argument list

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

Presentation Manager: A pointer to the process information block is also returned as an unsigned long value.

Starting and Stopping

Use these members to start or stop the current process.

exit

Ends the current thread of execution by calling ICurrentThread::exit (p. 448). If the current thread is thread 1, the process is ended.

```
virtual ICurrentApplication&  
    exit();
```

run

Starts the processing of events in the current process by calling
ICurrentThread::processMsgs (p. 452).

```
virtual ICurrentApplication&
    run();
```

Inherited Public Functions

IApplication		
adjustPriority	asString	currentPID
asDebugInfo	current	id

IVBase		
asDebugInfo	asString	

IBase		
asDebugInfo	messageFile	setMessageFile
asString	messageText	version

Protected Functions

Constructors

The constructors of this class are protected to ensure that the static function
IApplication::current (p. 423) returns the correct reference to the only object of this
class. Also, the destructor is protected.

ICurrentApplication

You can only construct objects of this class with the default constructor, which does
not require any arguments.

```
ICurrentApplication();
```

~ICurrentApplication

```
virtual
    ~ICurrentApplication();
```

Process Information

Use these members to access process information.

pib

Returns a pointer to the current process' process information block.

```
struct pib_s&  
pib();
```

Inherited Protected Functions

IApplication		
setId		

Inherited Protected Data

IBase		
recoverable	unrecoverable	

Chapter 64. ICurrentThread

Derivation

```

IBase
  IVBase
    IThread
      ICurrentThread
  
```

Inherited By

None.

Header File

ithread.hpp

Members

Member	Page
Constructor	455
anchorBlock	450
appContext	451
appShell	451
exit	448
handle	448
id	448
initializeGUI	451
isGUIInitialized	452
isTopLevelShell	448
isXErrorCodeAvailable	449
messageQueue	452
processMsgs	452
remainingStack	449
setTopLevelShell	453
setXErrorCode	453
sleep	449
startedThread	455
suspend	454
terminateGUI	452
waitFor	449
waitForAllThreads	450
waitForAnyThread	450
XerrorCode	454

The ICurrentThread class represents the current thread of execution. An object of this class contains information that the Application Support Class Library maintains for the current thread of execution. You are limited to a single object of this class.

This class provides functions that you can only apply to the current thread of execution. To obtain a reference to the object, use the static function IThread::current (p. 548).

Public Functions

Current Thread Information

Use these members to access general information on the current thread.

handle

Returns the thread handle for the current thread.

```
virtual IThreadHandle  
    handle() const;
```

Presentation Manager: The thread handle and thread ID are identical.

Windows: You must use the thread handle for the operating system thread and synchronization functions.

id

Obtains the current thread's identifier (ID).

```
virtual IThreadId  
    id() const;
```

Motif: The AIX release of the Application Support Class Library supports only one thread. Therefore, this function returns the value 1.

Current Thread Support

Use these members to enable thread operations that only apply to the current thread.

exit

Ends the current thread with the specified return value.

```
virtual void  
    exit( unsigned long returnCode);
```

returnCode

Unsigned long value that specifies the return value for termination.

isTopLevelShell

Determines if the application shell window is completely initialized. If it is, true is returned.

```
virtual Boolean  
    isTopLevelShell() const;
```

OS/390: This member function is not supported.

isXErrorCodeAvailable

Determines if an X Library error code is available for querying. If one is available, true is returned. This function uses XErrorCode to retrieve the error code, which resets this flag.

```
virtual Boolean
    isXErrorCodeAvailable() const;
```

remainingStack

Obtains an approximation of the number of available bytes that remain on the stack. If the stack is not initialized, 0 is returned.

```
virtual unsigned long
    remainingStack() const;
```

Windows: This function always returns 1000 because an approximation cannot be obtained.

sleep

Suspends the thread for a specified number of milliseconds.

```
virtual ICurrentThread&
    sleep( unsigned long milliseconds);
```

milliseconds

Unsigned long value that specifies the number of milliseconds to sleep.

Motif: The granularity for sleeping on AIX is in seconds. The number of milliseconds is divided by 1000 to determine the number of seconds the process will sleep. If the number of milliseconds is less than 1 second, the process will not sleep.

waitFor

Allows the current thread to wait for a specified thread to end.

```
virtual ICurrentThread&
    waitFor( const IThread& anotherThread);
```

anotherThread

Reference to an IThread object that represents the specified thread.

Windows (Win32s): This member function has no effect.

Motif: The AIX release of the Application Support Class Library supports a single-threaded environment; therefore, there are no other threads to wait for. This function immediately returns a reference to the ICurrentThread object.

Exceptions	
IInvalidParameter	The current thread did not wait for the specified thread to end. The specified thread is the current thread, which cannot wait on itself.
IAccessError	The current thread did not wait for the specified thread to end. The specified thread may have been invalid or an interrupt may have occurred.

waitForAllThreads

Allows the current thread to wait for all secondary threads to end.

```
virtual ICurrentThread&
    waitForAllThreads();
```

OS/390: This member function is not supported. An exception of type `invalidRequest` will result if this member is called.

Windows (Win32s): This member function has no effect.

Motif: The AIX release of the Application Support Class Library supports a single-threaded environment; therefore, there are no other threads to wait for. This function immediately returns a reference to the `ICurrentThread` object.

Exceptions	
<code>InvalidParameter</code>	The current thread did not wait for all secondary threads to end. The current thread is not the primary thread.

waitForAnyThread

Allows the current thread to wait for the first termination to occur of any other threads in the current process. The terminated thread's identifier (ID) is returned.

```
virtual IThreadId
    waitForAnyThread();
```

OS/390: This member function is not supported. An exception of type `invalidRequest` will result if this member is called.

Windows (Win32s): This member function has no effect.

Motif: The AIX release of the Application Support Class Library supports a single-threaded environment; therefore, there are no other threads to wait for. This function immediately returns a reference to the `ICurrentThread` object.

Exceptions	
<code>IAccessError</code>	The current thread did not wait for the next thread in the process to terminate. No thread terminated or an interrupt may have occurred.

Graphical User Interface (GUI) Support

Use these members to support GUI activities on the current thread.

anchorBlock

Obtains the anchor block handle for the current thread. If the anchor block handle is not initialized, 0 is returned.

```
virtual IAnchorBlockHandle
    anchorBlock() const;
```

OS/390: This member function is not supported.

Windows: This function always returns 1 because Windows has no concept of an anchor block.

appContext

Obtains the application context handle for the current thread. If the application context handle is not initialized, 0 is returned.

```
virtual IContextHandle  
    appContext() const;
```

OS/390: This member function is not supported.

appShell

Obtains the application shell handle for the current thread. If the application shell handle is not initialized, 0 is returned.

```
virtual IWindowHandle  
    appShell() const;
```

OS/390: This member function is not supported.

initializeGUI

Initializes the graphical user interface (GUI) environment on the current thread.

```
virtual void  
    initializeGUI( long queueSize = 30);
```

queueSize

Long value that represents the queue size for the GUI environment. The Application Support Class Library default queue size is 30.

OS/390: This member function is not supported.

Presentation Manager: Sets up a Presentation Manager environment (anchor block and message queue).

Windows: The *queueSize* parameter has no effect because you can neither create a message queue nor specify a queue size under Windows.

Windows (Win32s): You can set the message queue size using this function. You must do so before the creation of any window classes, otherwise the function has no effect. It is recommended that the message queue size be set before calling any Windows functions.

If you do not call this function the queue size is set to 96, the recommended message queue size for applications using OLE. The Application Support Class Library drag and drop support uses OLE.

Motif: Initializes the X Library environment by calling XtAppInitialize to establish a connection to the display. This function also creates an application context and an initial application shell.

Exceptions	
IAccessError	The graphical user interface (GUI) was not initialized. The window initialization failed.
IAccessError	The graphical user interface (GUI) was not initialized. The message queue creation failed.

ICurrentThread

isGUIInitialized

Determines if the graphical user interface (GUI) environment is active for this thread. If the GUI is active, true is returned.

```
virtual Boolean  
    isGUIInitialized() const;
```

OS/390: This member function is not supported.

messageQueue

Obtains the message queue handle for the current thread. A 0 is returned if the handle is not initialized or is invalid.

```
virtual IMessageQueueHandle  
    messageQueue();
```

Use this version of the function if you want to reset an invalid message queue handle to 0. Do this if the handle is determined to be invalid.

OS/390: This member function is not supported.

Windows: Returns the current thread's identifier.

```
virtual IMessageQueueHandle  
    messageQueue() const;
```

Use this version of the function if you want the invalid message queue handle to remain invalid.

OS/390: This member function is not supported.

processMsgs

Gets messages from the message queue and dispatches them to the appropriate handlers.

```
virtual void  
    processMsgs();
```

OS/390: This member function is not supported.

Presentation Manager: Messages are dispatched until the WM_QUIT message is received.

Motif: ICurrentThread dispatches messages until it receives the client message WM_QUIT or the application is terminated through the Motif Window Manager.

terminateGUI

Terminates the graphical user interface (GUI) environment. If you start a thread with IThread::autoInitGUI (p. 540) and the function returns true, this function is called automatically by the Application Support Class Library once IThreadFn::run (p. 560) is completed.

If you do the following, the Application Support Class Library does not call this function automatically, and it is your responsibility to do so:

- Start the thread with IThread::autoInitGUI and the function returns false
- Stop the thread using IThread::stop (p. 548)
- Stop the thread using ICurrentThread::exit (p. 448)

Note: If you do not call this function and it is your responsibility to do so, you can tie up system resources until your application ends.

```
virtual void
    terminateGUI();
```

OS/390: This member function is not supported.

Presentation Manager: Before calling the WinTerminate API, the application must destroy (that is delete) all windows and the message queue. This function destroys the message queue, but it is still the application's responsibility to destroy all of the window objects for the thread. If it does not, the return value of WinTerminate and any subsequent calls is indeterminate.

Windows: This function has no effect because the message queue is automatically destroyed by Windows.

Implementation

These members provide utilities used to implement this class. They are used by the Application Support Class Library.

setTopLevelShell

Called when the application shell window has been initialized. The Application Support Class Library calls this function to prevent multiple initializations of the graphical user interface.

Note: A user of the Application Support Class Library should never call this member function.

```
virtual void
    setTopLevelShell() const;
```

OS/390: This member function is not supported.

setXErrorCode

Sets the error code that the last call to an X Library function reported. Typically, you use this function to save the error code detected by the handler set by XSetErrorHandler in IThread. The error handler is set by IThread so that the error codes are maintained on a per thread basis.

Note: A user of the Application Support Class Library should never call this member function.

```
virtual void
    setXErrorCode( int errorCode);
```

errorCode

Integer value that specifies the error code to set.

OS/390: This member function is not supported.

XerrorCode

Returns the X Library error code set by the member function `ICurrentThread::setXerrorCode` (p. 453). You can retrieve the error code until a new X Library error code is set by `ICurrentThread::setXerrorCode`, but the flag indicating that an error code is available is reset on the first invocation. The class `IXLibErrorInfo` uses this function to obtain information about the last X Library error code detected.

Note: A user of the Application Support Class Library should never call this member function.

```
virtual int
    XerrorCode() const;
```

OS/390: This member function is not supported.

Suspending Threads

Use these members to suspend your current thread of execution if the thread is a non-GUI thread. Use `IThread::resume` (p. 546) to resume execution of the thread.

suspend

Suspends the current thread of execution only if the graphical user interface (GUI) is not initialized for the thread. Suspending a GUI-initialized thread causes your application to hang.

```
virtual void
    suspend();
```

OS/390: This member function is not supported. An exception of type `InvalidRequest` will result if this member is called.

Windows (Win32s): This function does not suspend the current thread.

Motif: You cannot suspend or resume AIX threads because the AIX release of the Application Support Class Library has a single-thread limitation.

Exceptions	
<code>InvalidRequest</code>	The current thread was not suspended. A GUI thread cannot be suspended, or the application will hang.

Inherited Public Functions

IThread		
<code>adjustPriority</code>	<code>id</code>	<code>setPriority</code>
<code>asDebugInfo</code>	<code>isStarted</code>	<code>setQueueSize</code>
<code>asString</code>	<code>messageQueue</code>	<code>setRelatedHandlesList</code>
<code>autoInitGUI</code>	<code>priorityClass</code>	<code>setStackSize</code>
<code>current</code>	<code>priorityLevel</code>	<code>setVariable</code>
<code>currentHandle</code>	<code>queueSize</code>	<code>setWindowList</code>
<code>currentId</code>	<code>relatedHandlesList</code>	<code>stackSize</code>
<code>defaultAutoInitGUI</code>	<code>resume</code>	<code>start</code>

IThread		
defaultQueueSize	setAutoInitGUI	stop
defaultStackSize	setDefaultAutoInitGUI	stopProcessingMsgs
dialogControls	setDefaultQueueSize	suspend
handle	setDefaultStackSize	variable

IVBase		
asDebugInfo	asString	

IBase		
asDebugInfo	messageFile	setMessageFile
asString	messageText	version

Protected Functions

Constructors

Use these protected constructors to construct objects of this class.

ICurrentThread

The constructor is protected to prevent the accidental creation of objects of this class. Creation of objects of this class is restricted to IThread::current (p. 548).

```
ICurrentThread();
```

Implementation

These members provide utilities used to implement this class. They are used by the Application Support Class Library.

startedThread

Returns a pointer to the object of the IStartedThread class that corresponds to the current thread.

```
virtual IStartedThread*
    startedThread() const;
```

Inherited Protected Functions

IThread		
newStartedThread	operator =	startedThread

Inherited Protected Data

IBase		
recoverable	unrecoverable	

Chapter 65. IEnumHandle

Derivation

Inherits from none.

Inherited By

None.

Header File

ibhandle.hpp

Members

Member	Page
Constructor	457
asDebugInfo	457
asString	458
asUnsigned	458
operator Value	458

The IEnumHandle class accesses lists of enumerated items. For example, an enumeration list can contain the handles of all the child windows for a given parent window.

Presentation Manager: IEnumHandle is an alias for the OS/2 Programmer's Toolkit type HENUM.

Motif: This class is implemented on AIX but has no functionality.

Public Functions

Constructors

You can construct objects of this class.

IEnumHandle

Constructs objects of this class from an enumeration handle (a value of type IHandle::Value), which defaults to 0.

```
IEnumHandle( Value henum = 0);
```

Diagnostics

asDebugInfo

Returns the handle as a string containing diagnostic information.

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

IEnumHandle

asString

Returns the handle as a string of form *nnn*.

```
IString  
    asString() const;
```

asUnsigned

Returns the handle value as an unsigned long value.

```
unsigned long  
    asUnsigned() const;
```

Operators

This group contains operators for this class.

operator Value

Returns the IHandle value.

```
operator Value() const;
```

Nested Type Definitions

Value

```
typedef void * Value;
```

Value

```
typedef unsigned long Value;
```


Chapter 66. IEventData

Derivation

```

IBase
IEventData

```

Inherited By

None.

Header File

ievtdata.hpp

Members

Member	Page
Constructor	460
asLong	462
asUnsignedLong	462
char1	461
char2	461
char3	461
char4	461
highHighByte	461
highLowByte	461
highNumber	461
lowHighByte	461
lowLowByte	462
lowNumber	462
number1	462
number2	462
operator char *	462
operator unsigned long	462

The IEventData class encapsulates the data of an event. IEventData is interchangeable with the following classes, which are actually aliases (that is, typedefs) for IEventData:

- IEventParameter1 (p. 465)
- IEventParameter2 (p. 467)
- IEventResult (p. 469)
- IHighEventParameter (p. 475)
- ILowEventParameter (p. 477)

Motif: Although the Application Support Class Library sometimes internally uses the IEventResult value returned by the application, this value is not returned to the X-Motif window system. X-Motif does not support returning any value. Particularly, it does not support the ability to prevent further processing of the event by others, as Presentation Manager does.

Public Functions

Constructors

You can construct and destruct objects of this class.

IEventData

```
IEventData();
```

Create an IEventData object with the event data set to 0.

```
IEventData( void* value);
```

Create an IEventData object from a pointer to a void. The event data is set to the specified void pointer.

```
IEventData( unsigned long value);
```

Create an IEventData object from an unsigned long integer value. The event data is set to the specified unsigned long integer.

```
IEventData( int value);
```

Create an IEventData object from an integer. The event data is set to the specified integer. This constructor accepts an uncasted value of 0.

```
IEventData( BooleanConstants value);
```

Create an IEventData object from a BooleanConstants value. This constructor accepts one of the two IBase::BooleanConstants values, true or false. The event data is set to the Boolean value.

```
IEventData( unsigned short lowValue,  
            unsigned short hiValue);
```

Create an IEventData object from two unsigned short integer values. The event data is set to an unsigned long integer whose two words are the two specified unsigned short integers.

```
IEventData( unsigned short lowValue,  
            char lowByte,  
            char hiByte);
```

Create an IEventData object using the specified unsigned short value and two characters. The event data is set to an unsigned long integer whose two words are the unsigned short integer in the low word and the two characters in the high word.

Contents

These members query and set the event data contained by objects of this class.

char1

Returns the character in the low byte of the event data's low word. This is the same as `lowLowByte` (p. 462).

```
char  
    char1() const;
```

char2

Returns the character in the high byte of the event data's low word. This is the same as `lowHighByte` (p. 461).

```
char  
    char2() const;
```

char3

Returns the character in the low byte of the event data's high word. This is the same as `highLowByte` (p. 461).

```
char  
    char3() const;
```

char4

Returns the character in the high byte of the event data's high word. This is the same as `highHighByte` (p. 461).

```
char  
    char4() const;
```

highHighByte

Returns the character in the high byte of the event data's high word. This is the same as `char4` (p. 461).

```
char  
    highHighByte() const;
```

highLowByte

Returns the character in the low byte of the event data's high word. This is the same as `char3` (p. 461).

```
char  
    highLowByte() const;
```

highNumber

Returns the event data's high word. This is the same as `number2` (p. 462).

```
unsigned short  
    highNumber() const;
```

lowHighByte

Returns the character in the high byte of the event data's low word. This is the same as `char2` (p. 461).

```
char  
    lowHighByte() const;
```

IEventData

lowLowByte

Returns the character in the low byte of the event data's low word. This is the same as char1 (p. 461).

```
char  
    lowLowByte() const;
```

lowNumber

Returns the event data's low word. This is the same as number1 (p. 462).

```
unsigned short  
    lowNumber() const;
```

number1

Returns the event data's low word. This is the same as lowNumber (p. 462).

```
unsigned short  
    number1() const;
```

number2

Returns the event data's high word. This is the same as highNumber (p. 461).

```
unsigned short  
    number2() const;
```

Conversion

These members convert an IEventData object to a four-byte value. You can either explicitly convert the object by calling a member function or casting, or allow the compiler to implicitly convert the object using an operator.

asLong

Returns the long event data value.

```
long  
    asLong() const;
```

asUnsignedLong

Returns the unsigned long event data value.

```
unsigned long  
    asUnsignedLong() const;
```

operator char *

Returns the event data as a pointer to a character.

```
operator char *() const;
```

operator unsigned long

Returns the event data as an unsigned long number.

```
operator unsigned long() const;
```

Inherited Public Functions

IBase		
asDebugInfo	messageFile	setMessageFile
asString	messageText	version

Inherited Protected Data

IBase		
recoverable	unrecoverable	

Chapter 67. IEventParameter1

Derivation

```

IBase
IEventParameter1

```

Inherited By

None.

Header File

ievtdat2.hpp

Portability Considerations: The IEventParameter1 class encapsulates the message parameter (MPARAM) and result (HRESULT) data of an event on OS/2 and Windows. In the OS/390 environment, the IEventParameter1 class is user-defined.

IEventParameter1 is an alias of the IEventData class. See IEventData (p. 459) for more information.

In addition, IEventParameter1 is interchangeable with the following classes:

IEventParameter2 (p. 467)
 IEventResult (p. 469)
 IHighEventParameter (p. 475)
 ILowEventParameter (p. 477)

Inherited Public Functions

IBase		
asDebugInfo	messageFile	setMessageFile
asString	messageText	version

Inherited Protected Data

IBase		
recoverable	unrecoverable	

Chapter 68. IEventParameter2

Derivation

```

IBase
IEventParameter2

```

Inherited By

None.

Header File

ievtdat2.hpp

Portability Considerations: The IEventParameter2 class encapsulates the message parameter (MPARAM) and result (HRESULT) data of an event on OS/2 and Windows. In the OS/390 environment, the IEventParameter2 class is user-defined.

IEventParameter2 is an alias of the IEventData class. See IEventData (p. 459) for more information.

In addition, IEventParameter2 is interchangeable with the following classes:

```

IEventParameter1 (p. 465)
IEventResult (p. 469)
IHighEventParameter (p. 475)
ILowEventParameter (p. 477)

```

Inherited Public Functions

IBase		
asDebugInfo	messageFile	setMessageFile
asString	messageText	version

Inherited Protected Data

IBase		
recoverable	unrecoverable	

Chapter 69. IEventResult

Derivation

```

IBase
IEventResult

```

Inherited By

None.

Header File

ievtdat2.hpp

Portability Considerations: The IEventResult class encapsulates the message parameter (MPARAM) and result (HRESULT) data of an event on OS/2 and Windows. In the OS/390 environment, the IEventResult class is user-defined.

IEventResult is an alias of the IEventData class. See IEventData (p. 459) for more information.

In addition, IEventResult is interchangeable with the following classes:

- IEventParameter1 (p. 465)
- IEventParameter2 (p. 467)
- IHighEventParameter (p. 475)
- ILowEventParameter (p. 477)

Inherited Public Functions

IBase		
asDebugInfo	messageFile	setMessageFile
asString	messageText	version

Inherited Protected Data

IBase		
recoverable	unrecoverable	

Chapter 70. IHandle

Derivation

```

IBase
IHandle

```

Header File

```
ibhandle.hpp
```

Members

Member	Page
Constructor	472
asDebugInfo	472
asString	472
asUnsigned	472
handle	473
operator Value	472

The IHandle class is a base class for all of the concrete handle-derived classes. You can create objects of this class, although such objects have limited utility. The Application Support Class Library APIs specify classes derived from IHandle.

The advantage of providing separate classes for each handle type is that they become distinct enough to prevent a user from passing the wrong handle type to a given function.

This base class manages the handle data member by defining the following:

- A constructor that accepts an unsigned long and stores the handle in the handle field.
- A user conversion operator as type IHandle. This operator permits IHandle objects to be used anywhere an IHandle is required (typically on windowing system or host operating system APIs).

There are numerous derived classes of IHandle representing each of the supported system handle types.

Public Functions

Constructors

You can construct objects of this class.

IHandle

IHandle

This constructor accepts an object of IHandle type Value as an argument. Typically this value originates in a system API call.

```
IHandle( Value value);
```

Diagnostics

Use these members to get and set the accessible attributes of objects of this class.

asDebugInfo

Returns the handle as a string containing diagnostic information.

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

asString

Returns the handle as a string of form *nnnn*.

```
IString  
asString() const;
```

asUnsigned

Returns the handle value as an unsigned long value.

```
unsigned long  
asUnsigned() const;
```

Operators

This group contains operators for this class.

operator Value

Returns the IHandle value.

```
operator Value() const;
```

Inherited Public Functions

IBase		
asDebugInfo	messageFile	setMessageFile
asString	messageText	version

Protected Data

Value

These members contain information about this class.

handle

This member is used to hold the handle value.

```
Value  
    handle;
```

Inherited Protected Data

IBase		
recoverable	unrecoverable	

Nested Type Definitions

Value

```
typedef unsigned long Value;
```


Chapter 71. IHighEventParameter

Derivation

```

IBase
IHighEventParameter

```

Inherited By

None.

Header File

ievtdat2.hpp

Portability Considerations: The IHighEventParameter class encapsulates the message parameter (MPARAM) and result (MRESULT) data of an event on OS/2 and Windows. In the OS/390 environment, the IHighEventParameter class is user-defined.

IHighEventParameter is an alias of the IEventData class. See IEventData (p. 459) for more information.

In addition, IHighEventParameter is interchangeable with the following classes:

- IEventParameter1 (p. 465)
- IEventParameter2 (p. 467)
- IEventResult (p. 469)
- ILowEventParameter (p. 477)

Inherited Public Functions

IBase		
asDebugInfo	messageFile	setMessageFile
asString	messageText	version

Inherited Protected Data

IBase		
recoverable	unrecoverable	

Chapter 72. ILowEventParameter

Derivation

```

IBase
  ILowEventParameter

```

Inherited By

None.

Header File

ievtdat2.hpp

Portability Considerations: The ILowEventParameter class encapsulates the message parameter (MPARAM) and result (MRESULT) data of an event on OS/2 and Windows. In the OS/390 environment, the ILowEventParameter class is user-defined.

ILowEventParameter is an alias of the IEventData class. See IEventData (p. 459) for more information.

In addition, ILowEventParameter is interchangeable with the following classes:

- IEventParameter1 (p. 465)
- IEventParameter2 (p. 467)
- IEventResult (p. 469)
- IHighEventParameter (p. 475)

Inherited Public Functions

IBase		
asDebugInfo	messageFile	setMessageFile
asString	messageText	version

Inherited Protected Data

IBase		
recoverable	unrecoverable	

Chapter 73. INotificationEvent

Derivation

IBase
INotificationEvent

Inherited By

None.

Header File

inotifev.hpp

Members

Member	Page
Constructor	479
eventData	480
hasNotifierAttrChanged	480
notificationId	480
notifier	481
observerData	481
operator =	480
setEventData	481
setNotifierAttrChanged	481
setObserverData	481
~INotificationEvent	480

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.

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 ( ));
```

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

INotificationEvent

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.

```
INotifier&  
    notifier() const;
```

observerData

Returns observer data that is added when the observer registers with the notifier object.

```
IEventData  
    observerData() const;
```

setEventData

Stores event data that is specific to a particular notification. The existence and type of the event data is documented with the notification IDs in the definition of the derived notifier class.

```
INotificationEvent&  
    setEventData( const IEventData& eventData);
```

setNotifierAttrChanged

Indicates that the notification event is a change in one of the notifier's attributes.

```
INotificationEvent&  
    setNotifierAttrChanged( Boolean notifierAttrchanged = true);
```

setObserverData

Stores observer data in the notification event. The observer provides this data when it registers with a notifier by calling the INotifier::addObserver protected member function.

```
INotificationEvent&  
    setObserverData( const IEventData& observerData);
```

Inherited Public Functions

IBase		
asDebugInfo	messageFile	setMessageFile
asString	messageText	version

Inherited Protected Data

IBase		
recoverable	unrecoverable	

Chapter 74. INotifier

Derivation

```

IBase
IVBase
INotifier

```

Inherited By

```

IStandardNotifier

```

Header File

```

inotify.hpp

```

Members

Member	Page
Constructor	484
addObserver	485
disableNotification	484
enableNotification	484
isEnabledForNotification	484
notifyObservers	485
observerList	486
removeAllObservers	486
removeObserver	486
~INotifier	484

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.

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

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 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.

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;
```

isEnabledForNotification

Returns true if a notifier can send notifications to its observers.

```
virtual Boolean  
isEnabledForNotification() const = 0;
```

Observer Notification

These members notify observers of a change in a notifier.

notifyObservers

Notifies all observers in a notifier's list of observers. Each observer receives a notification event containing the identity of the notifier, the notification ID, and any optional data provided by the specific notifier object.

Note: A public and a protected version of notifyObservers are provided for convenience. The protected version does not require the caller to construct an INotificationEvent to call it. In this case, the construction of the INotificationEvent occurs in the code of the protected notifyObservers function.

```
virtual INotifier&
    notifyObservers( const INotificationEvent& event) = 0;
```

Inherited Public Functions

IVBase		
asDebugInfo	asString	

IBase		
asDebugInfo	messageFile	setMessageFile
asString	messageText	version

Protected Functions

Observer Addition and Removal

IObserver objects use these members to add and remove themselves from the notifier's collection.

addObserver

Adds an observer to the notifier's list of observers.

```
virtual INotifier&
    addObserver( IObserver& observer,
                 const IEventData& userData) = 0;
```

INotifier

observerList

Returns the list of observers. If the observer list does not exist, the derived notifier class must create it before calling this member function.

```
virtual IObserverList&
observerList() const = 0;
```

removeAllObservers

Removes all observers from the notifier's list of observers.

```
virtual INotifier&
removeAllObservers() = 0;
```

removeObserver

Removes an observer from the notifier's list of observers.

```
virtual INotifier&
removeObserver( IObserver& observer) = 0;
```

Observer Notification

These members notify observers of a change in a notifier.

notifyObservers

Notifies all observers in a notifier's list of observers. Each observer receives a notification event containing the identity of the notifier, the notification ID, and any optional data provided by the specific notifier object.

Note: A public and a protected version of notifyObservers are provided for convenience. The protected version does not require the caller to construct an INotificationEvent to call it. In this case, the construction of the INotificationEvent occurs in the code of the protected notifyObservers function.

```
virtual INotifier&
notifyObservers( const INotificationId& id) = 0;
```

Inherited Protected Data

IBase		
recoverable	unrecoverable	

Chapter 75. IObserver

Derivation

```

IBase
IVBase
IObserver

```

Inherited By

None.

Header File

iobserver.hpp

Members

Member	Page
Constructor	488
dispatchNotificationEvent	489
handleNotificationsFor	488
stopHandlingNotificationsFor	488
~IObserver	487

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.

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	

Chapter 76. IObserverList

Derivation

```

IBase
IVBase
IObserverList

```

Inherited By

None.

Header File

iobslist.hpp

Members

Member	Page
Constructor	491
add	492
elementAt	492
isEmpty	492
notifyObservers	493
numberOfElements	492
remove	492
removeAll	493
removeAt	493
~IObserverList	492

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

~IObserverList

```
virtual  
~IObserverList();
```

Observer Addition and Removal

Use these members to add, remove, and find IObserver objects in the observer list's collection.

add

Adds an observer to the end of the list.

```
virtual Boolean  
add( IObserver& observer,  
      void* userData);
```

elementAt

Returns an observer from the list using the specified cursor object.

```
virtual IObserver&  
elementAt( const Cursor& cursor) const;
```

isEmpty

Returns true if there are no observers in the list.

```
Boolean  
isEmpty() const;
```

numberOfElements

Returns the number of observers in the list.

```
unsigned long  
numberOfElements() const;
```

remove

Removes the specified observer from the list.

```
virtual IObserverList&  
remove( const IObserver& observer);
```

removeAll

Removes all observers from the list.

```
virtual IObserverList&
removeAll();
```

removeAt

Removes an observer at the specified cursor location from the list.

```
virtual IObserverList&
removeAt( const Cursor& cursor);
```

Observer Notification

These members notify observers of a change in a notifier.

notifyObservers

Traverses the list of observers and calls each member's dispatchNotificationEvent function passing a specified notification event object.

```
IObserverList&
notifyObservers( const INotificationEvent& event);
```

Inherited Public Functions

IVBase		
asDebugInfo	asString	

IBase		
asDebugInfo	messageFile	setMessageFile
asString	messageText	version

Inherited Protected Data

IBase		
recoverable	unrecoverable	

Nested Classes

IObserverList contains the following nested classes:

IObserverList::Cursor (see page 495)

Chapter 77. IObserverList::Cursor

Derivation

```
IBase
IVBase
IObserverList::Cursor
```

Inherited By

None.

Header File

iobslist.hpp

Members

Member	Page
Constructor	495
Cursor	495
invalidate	496
isValid	496
setToFirst	496
setToLast	496
setToNext	496
setToPrevious	496
~Cursor	495

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.

```
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

Inherited Protected Data

IBase		
recoverable	unrecoverable	

Chapter 78. IPrivateResource

Derivation

```
IBase
  IVBase
    IResource
      IPrivateResource
```

Inherited By

None.

Header File

ireslock.hpp

Members

Member	Page
Constructor	500
~IPrivateResource	500

The IPrivateResource class defines a resource that is used within a single process.

IResource (p. 515) and its inherited classes represent resources in the user's problem domain. Objects of these classes name a particular resource. For example, a user wants to serialize access to the data in a window list collection. The user creates an IResource inherited class to represent the window list and then uses IResourceLock (p. 519) (preferred) or a function of IResource itself to lock and unlock the resource to serialize the access. You can use this class either alone (lock and unlock) or in combination with an IResourceLock, whose constructor locks the resource and whose destructor unlocks the resource. If you need the lock only for a block of code, use IResourceLock. Use static objects because you only need a single object of a particular IResource.

The static object pointer discussions exist because there is no language-defined way to ensure that static objects are constructed when needed. Also, they are always constructed even if they are never needed. Combining a static pointer to an object with a static function to reference the object and then creating the object, if necessary, defers creating the object until it is needed.

Use this class for resources that are limited to the same process (as opposed to a public resource that is used by more than one process via its name).

You can use this class as a static key to serialize access to a private resource. You can also use this class as a mechanism to ensure that the static resource is constructed prior to being used.

Some basic guidelines include the following:

- Use a static pointer to the resource rather than a static object.
- Always access the resource using a static function rather than accessing it directly with the static pointer.

IPrivateResource

- When the resource is accessed through the static function, allocate it if the static pointer is 0.
- Provide a new class that represents the static pointers for a particular class or component. This new class does not require a constructor, but it does require a destructor that destroys the static objects used by the component.

Windows (Win32s): Resource locks are not supported. Instances of the IPrivateResource class can be created and the member functions can be called, but no system resource locks are obtained.

Public Functions

Constructors

Use these members to construct and destruct objects of this class.

IPrivateResource

Provides the default constructor. The only way to construct objects of this class is with this, the default constructor.

```
IPrivateResource();
```

Windows (Win32s): Resource locks are not supported. No resources are created when calling this constructor.

Exceptions	
IAccessError	The private resource object was not created. The memory may be exhausted or the semaphore handle limit on your platform may be exceeded.

~IPrivateResource

```
virtual  
~IPrivateResource();
```

Windows (Win32s): Resource locks are not supported. No resources are released when calling this destructor.

Exceptions	
IAccessError	The private resource object was not deleted. The semaphore handle may be invalid.

Inherited Public Functions

IResource		
lock	unlock	

IVBase		
asDebugInfo	asString	

IBase		
asDebugInfo	messageFile	setMessageFile
asString	messageText	version

Inherited Protected Functions

IResource		
handle		

Inherited Protected Data

IBase		
recoverable	unrecoverable	

Chapter 79. IPrivateSemaphoreHandle

Derivation

Inherits from none.

Inherited By

None.

Header File

ibhandle.hpp

Members

Member	Page
Constructor	503
asDebugInfo	504
asString	504
asUnsigned	504
operator =	504
operator Value	504
~IPrivateSemaphoreHandle	503

The IPrivateSemaphoreHandle class accesses semaphores. An example of a semaphore is a flag in a multiple-user application that prevents simultaneous access to a file.

Public Functions

Constructors

You can construct objects of this class.

IPrivateSemaphoreHandle

You can construct objects of this class from a semaphore handle (a value of type IHandle::Value), which defaults to 0.

```
IPrivateSemaphoreHandle( Value privSemId = 0);
```

~IPrivateSemaphoreHandle

```
~IPrivateSemaphoreHandle();
```

Diagnostics

Use these members to get and set the accessible attributes of objects of this class.

IPrivateSemaphoreHandle

asDebugInfo

Returns the handle as a string containing diagnostic information.

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

asString

Returns the handle as a string of form *nnn*.

```
IString  
    asString() const;
```

Note: For OS/390, the returned handle is not of form *nnn*. it is a string of hex numbers.

asUnsigned

Returns the handle value as an unsigned long value.

```
unsigned long  
    asUnsigned() const;
```

Operators

This group contains operators for this class.

operator =

The assignment operator. You can assign one IPrivateSemaphoreHandle to another or you can assign a pointer to the referenced type.

```
IPrivateSemaphoreHandle&  
    operator =( pthread_mutex_t* privSemId );
```

operator Value

Returns the IHandle value.

```
operator Value() const;
```

Nested Type Definitions

Value

```
typedef void * Value;
```

Value

```
typedef pthread_mutex_t * Value;
```

Value

```
typedef unsigned long Value;
```

Chapter 80. IProcessId

Derivation

Inherits from none.

Inherited By

None.

Header File

ibhandle.hpp

Members

Member	Page
Constructor	505
asDebugInfo	505
asString	506
asUnsigned	506
operator Value	506

The IProcessId class accesses numeric identifiers for processes.

Presentation Manager: IProcessId is an alias for the OS/2 Programmer's Toolkit type PID.

Motif: IProcessId is an alias for the system type pid_t.

Public Functions

Constructors

You can construct objects of this class.

IProcessId

Constructs objects of this class from a process ID (a value of type IHandle::Value), which defaults to 0.

```
IProcessId( Value pid = 0);
```

Diagnostics

asDebugInfo

Returns the handle as a string containing diagnostic information.

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

IProcessId

asString

Returns the handle as a string of form *nnn*.

```
IString  
    asString() const;
```

asUnsigned

Returns the handle value as an unsigned long value.

```
unsigned long  
    asUnsigned() const;
```

Operators

This group contains operators for this class.

operator Value

Returns the IHandle value.

```
operator Value() const;
```

Nested Type Definitions

Value

```
typedef void * Value;
```

Value

```
typedef unsigned long Value;
```


Chapter 81. IRefCounted

Derivation

```

IBase
  IVBase
    IRefCounted

```

Inherited By

```

IDMItem
IDMOperation
IStringGeneratorFn
IThreadFn
ITimerFn

```

Header File

```
irefcnt.hpp
```

Members

Member	Page
Constructor	508
addRef	508
removeRef	508
useCount	508
~IRefCounted	509

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. 508) 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. 511), where T is a derived class of IRefCounted.

Public Functions

Reference Counting

Use these members to manage the object's reference count.

IRefCounted

addRef

Adds a reference to the referred-to object.

```
virtual void  
addRef();
```

removeRef

Removes a reference to the referred-to object. When the reference count goes to 0, this function deletes the referred-to object.

```
virtual void  
removeRef();
```

useCount

Returns the use count for the referred-to object.

```
unsigned  
useCount() const;
```

Inherited Public Functions

IVBase		
asDebugInfo	asString	

IBase		
asDebugInfo	messageFile	setMessageFile
asString	messageText	version

Protected Functions

Constructors

These members are protected.

IRefCounted

```
IRefCounted();
```

~IRefCounted

```
~IRefCounted();
```

Inherited Protected Data

IBase		
recoverable	unrecoverable	

Chapter 82. IReference

Derivation

```
IBase
IReference
```

Inherited By

None.

Header File

irefcnt.hpp

Members

Member	Page
Constructor	512
operator *	513
operator ->	513
operator =	512
operator T *	513
~IReference	512

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. 507). It must have the member functions IRefCounted::addRef (p. 508) and IRefCounted::removeRef (p. 508) 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.
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

```
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).

```
IReference( const IReference < T >& source );
```

You can construct objects of this class by using this copy constructor, which the library provides to ensure that the reference counts for both the source and target referents are maintained properly.

operator =

The assignment operator. You can assign one IReference to another or you can assign a pointer to the referenced type.

```
IReference < T >&  
operator =( const IReference < T >& source );
```

```
IReference < T >&  
operator =( T* p );
```

~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	

Chapter 83. IResource

Derivation

```
IBase
IVBase
IResource
```

Inherited By

```
IPrivateResource
ISharedResource
```

Header File

```
ireslock.hpp
```

Members

Member	Page
Constructor	515
lock	516
unlock	516
~IResource	515

The IResource class is the abstract base resource class. Use a derived class to identify a serially reusable resource. These resources can be limited to the current process, as described by the IPrivateResource (p. 499) class, or they can be shared across multiple processes, as described by ISharedResource (p. 523) class.

Public Functions

Constructors

You cannot construct or destruct objects of this class. Derive a new class from this class to identify a serially reusable resource if the derived Application Support Class Library classes do not meet your needs.

IResource

This function is the default constructor for this class.

```
IResource();
```

~IResource

```
virtual
~IResource();
```

Resource Locking

Use resource-locking members to acquire or release a serialized access to this resource.

lock

Acquires serialized access to the resource, preventing other threads of execution from accessing it. You can specify the value for *timeout* in milliseconds with -1 indicating an indefinite wait and 0 requesting an immediate return.

```
virtual IResource&
    lock( long timeout = - 1);
```

timeout Long value that represents in milliseconds how long to wait for this resource. The default value, -1, is an indefinite wait.

OS/390: Timeout values are not supported in a POSIX environment. The default value of -1 is assumed. Resource locks are only supported when the kernel is available and active. In a non-OS/390 UNIX environment, the lock() function has no effect.

Windows (Win32s): Resource locks are not supported. No resources are acquired and locked when calling this member function.

Motif: The X timer services used to implement the resource lock timeout have a resolution of one second. The lock function rounds up the specified timeout value to the next one-second interval.

unlock

Releases access to the resource, allowing other threads of execution to access it.

```
virtual IResource&
    unlock();
```

OS/390: Resource locks are only supported when the kernel is available and active. In a non-OS/390 UNIX environment, the unlock() member function has no effect.

Windows (Win32s): Resource locks are not supported. No resources are unlocked and released when calling this member function.

Inherited Public Functions

IVBase		
asDebugInfo	asString	

IBase		
asDebugInfo	messageFile	setMessageFile
asString	messageText	version

Inherited Protected Data

IBase		
recoverable	unrecoverable	

Chapter 84. IResourceLock

Derivation

```

IBase
IVBase
IResourceLock

```

Inherited By

None.

Header File

ireslock.hpp

Members

Member	Page
Constructor	519
clearLock	520
setLock	521
~IResourceLock	520

The IResourceLock class locks a resource for a specified period of time.

A simple yet effective use of this class is to declare a local object of this class that is in scope for the period of time that the resource needs to be locked. The lock is automatically removed when the block of code is exited, thereby forcing the object out of scope.

If the specified period of time is reached before the resource can be acquired, an IResourceExhausted (p. 369) exception is thrown.

Windows (Win32s): Win32s is a single threaded system. As a result, resource locks are not supported. Instances of the IResourceLock class can be created and the member functions can be called, but no system resource locks are obtained.

Public Functions

Constructors

Use these members to construct and destruct objects of this class.

IResourceLock

The only way to construct objects of this class is with this constructor. You specify a reference to an IResource (p. 515) and an optional value for the *timeOut* parameter. You can specify *timeOut* in milliseconds with -1 indicating an indefinite wait and 0 requesting an immediate return.

```

IResourceLock( IResource& resource,
               long timeOut = - 1);

```

IResourceLock

resource Reference to the resource being locked.
timeOut Long value that represents in milliseconds how long to wait for the lock request. The default value, -1, is an indefinite wait.

OS/390: Timeout values are not supported in a POSIX environment. Resource locks are only supported when the kernel is available and active.

Windows (Win32s): Resource locks are not supported. No resources are created when calling this constructor.

Motif: The AIX timer services used to implement the resource lock *timeOut* have a resolution of one second. The timeout values you provide are rounded up to the next one-second interval.

~IResourceLock

```
virtual  
    ~IResourceLock();
```

OS/390: Resource locks are only supported when the kernel is available and active.

Windows (Win32s): Resource locks are not supported. No resources are released when calling this destructor.

Inherited Public Functions

IVBase		
asDebugInfo	asString	

IBase		
asDebugInfo	messageFile	setMessageFile
asString	messageText	version

Protected Functions

Resource Locking

Use these members to acquire or release a lock on the resource.

clearLock

Releases the resource.

```
virtual IResourceLock&  
    clearLock();
```

OS/390: Resource locks are only supported when the kernel is available and active.

Windows (Win32s): Resource locks are not supported. No resources are cleared when calling this member function.

Exceptions	
IAccessError	The resource object was not released. The semaphore handle may be invalid, there may be too many release requests, the release request was interrupted, or the process which owns the semaphore died.

setLock

Acquires the resource. You can specify the value for *timeOut* in milliseconds with -1, indicating an indefinite wait, and 0, requesting an immediate return.

The constructor calls this function to lock the resource; IResourceLock::clearLock (p. 520) can then be called to clear the lock, and this function can be called to reset it. This is not the normal use of this class. Normally, you use the constructor and scope it to the instruction that requires serialization.

```
virtual IResourceLock&
    setLock( long timeOut = - 1);
```

timeOut Long value that represents in milliseconds how long to wait for the lock request. The default value, -1, is an indefinite wait.

OS/390: Timeout values are not supported in a POSIX environment. Resource locks are only supported when the kernel is available and active.

Windows (Win32s): Resource locks are not supported. No resources are acquired when calling this member function.

Motif: The AIX timer services used to implement the resource lock *timeOut* have a resolution of one second. This function rounds up the specified timeout value to the next one-second interval.

Exceptions	
IOutOfSystemResource	The resource object was not acquired. The available semaphores may be exhausted because the request for a lock timed out.
IAccessError	The resource object was not acquired. The semaphore handle may be invalid, there may be too many lock requests, the lock request was interrupted, or the process which owns the semaphore died.

Inherited Protected Data

IBase		
recoverable	unrecoverable	

Chapter 85. ISharedResource

Derivation

```
IBase
  IVBase
    IResource
      ISharedResource
```

Inherited By

None.

Header File

ireslock.hpp

Members

Member	Page
Constructor	524
keyName	525
~ISharedResource	524

The ISharedResource class defines a resource that can be shared across multiple processes. Use the name you provide on construction to distinguish between shared resources.

| OS/390: The ISharedResource class is only available if the kernel is available and
| active. In a non-OS/390 UNIX environment, any attempts to instantiate this class
| will result in an exception of type InvalidRequest being raised.

Windows (Win32s): Resource locks are not supported. Instances of the ISharedResource class can be created and the member functions can be called, but no system resource locks are obtained.

Motif: You might use ISharedResource objects to serialize resource access with applications that were not written using the Application Support Class Library. To make this work correctly, your non-Application Support Class Library application needs to use the AIX semaphore object in a manner consistent with the Application Support Class Library. To help you understand how the semaphore object is used by the Application Support Class Library, the basic interaction with the system's subroutines is described below.

The AIX semop subroutine is used for the semaphore lock and release operations. The ISharedResource constructor uses the semget subroutine to allocate a set containing two semaphores. The first semaphore in this set represents the state of the lock. The constructor initializes the semval field to free (value of 0). A lock operation increments the value, while an unlock operation decrements the value.

Recursive lock operations by the same process are allowed. In this case, the resource remains owned by the process until an equal number of unlock operations are done or the object is deleted. The second semaphore in the set is used as a reference counter to indicate the number of ISharedResource objects that are

referencing this semaphore set. The constructor increments this counter and the destructor decrements it. When the counter reaches 0, the destructor calls the `semctl` subroutine to delete the semaphore set.

The constructor determines the semaphore identifier to use in the `semget` subroutine by using the `ftok` system subroutine. If the parameter provided on the constructor begins with a slash (/) or is an existing file, the parameter is used as a file name for the `ftok` call and is the basis of the semaphore identifier token. If the parameter is not the name of an existing file, the constructor interprets the string as the name of a file in the `/tmp` directory. If this file does not exist, it is created. You can use `keyName` to obtain the file name used on the `ftok` subroutine call. The constructor always uses a value of 1 for the second parameter of `ftok`.

Portability Considerations: Use a `keyName` value that is a valid file name in all of the target environments.

Public Functions

Constructors

Use these members to construct and destruct objects of this class.

ISharedResource

You can only construct objects of this class by providing a name that uniquely identifies the resource to be shared.

```
ISharedResource( const char* keyName);
```

keyName Pointer to the name that uniquely identifies the resource.

Windows (Win32s): Resource locks are not supported. No resources are created when calling this constructor.

Exceptions	
<code>IAccessError</code>	The shared resource object was not created. The memory may be exhausted, the handle limit on your platform may be exceeded, or the <i>keyName</i> may be invalid.

~ISharedResource

```
virtual  
~ISharedResource();
```

Windows (Win32s): Resource locks are not supported. No resources are released when calling this destructor.

Exceptions	
<code>IAccessError</code>	The shared resource object was not deleted. The semaphore handle may be invalid.

Resource Information

Use these members to access shared resource information.

keyName

Returns the name used to identify this shared resource.

```
NSString  
keyName() const;
```

Inherited Public Functions

IResource		
lock	unlock	

IVBase		
asDebugInfo	asString	

IBase		
asDebugInfo	messageFile	setMessageFile
asString	messageText	version

Inherited Protected Functions

IResource		
handle		

Inherited Protected Data

IBase		
recoverable	unrecoverable	

Chapter 86. ISharedSemaphoreHandle

Derivation

Inherits from none.

Inherited By

None.

Header File

ibhandle.hpp

Members

Member	Page
Constructor	527
asDebugInfo	527
asString	528
asUnsigned	528
operator Value	528

The ISharedSemaphoreHandle class accesses semaphores. An example of a semaphore is a flag in a multiple-user application that prevents simultaneous access to a file.

Public Functions

Constructors

You can construct objects of this class.

ISharedSemaphoreHandle

You can construct objects of this class from a semaphore handle (a value of type IHandle::Value), which defaults to 0.

```
ISharedSemaphoreHandle( Value hsem = 0 );
```

Diagnostics

Use these members to get and set the accessible attributes of objects of this class.

asDebugInfo

Returns the handle as a string containing diagnostic information.

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

ISharedSemaphoreHandle

asString

Returns the handle as a string of form *nnn*.

```
IString  
    asString() const;
```

asUnsigned

Returns the handle value as an unsigned long value.

```
unsigned long  
    asUnsigned() const;
```

Operators

This group contains operators for this class.

operator Value

Returns the IHandle value.

```
operator Value() const;
```

Nested Type Definitions

Value

```
typedef void * Value;
```

Value

```
typedef unsigned long Value;
```

Chapter 87. IStandardNotifier

Derivation

IBase
IVBase
INotifier
IStandardNotifier

Inherited By

None.

Header File

istdntfy.hpp

Members

Member	Page
Constructor	530
addObserver	532
deleteId	533
disableNotification	530
enableNotification	530
isEnabledForNotification	531
notifyObservers	531
observerList	532
operator =	530
removeAllObservers	532
removeObserver	532
~IStandardNotifier	530

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

Public Functions

Constructors

You can construct, destruct, assign, and copy objects of this class.

IStandardNotifier

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();
```

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. 479) to call it. In this case, the construction of the INotificationEvent (p. 479) object occurs in the code of the protected notifyObservers function.

```
virtual IStandardNotifier&
    notifyObservers( const INotificationEvent& anEvent);
```

Inherited Public Functions

INotifier		
disableNotification	enableNotification	isEnabledForNotification

IVBase		
asDebugInfo	asString	

IBase		
asDebugInfo	messageFile	setMessageFile
asString	messageText	version

Protected Functions**Observer Addition and Removal**

Use these members to manage the collection of observers maintained by the notifier.

IStandardNotifier

addObserver

Adds an observer to the object's list of observers.

```
virtual IStandardNotifier&
addObserver( IObserver& observer,
            const IEventData& userData = IEventData ( 0 ));
```

observerList

Returns the list of IObservers. The list is created if it does not exist.

```
virtual IObservableList&
observerList() const;
```

removeAllObservers

Removes all observers from the object's observer list.

```
virtual IStandardNotifier&
removeAllObservers();
```

removeObserver

Removes an observer from the object's observer list.

```
virtual IStandardNotifier&
removeObserver( IObservable& 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. 479) to call it. In this case, the construction of the INotificationEvent (p. 479) object occurs in the code of the protected notifyObservers function.

```
virtual IStandardNotifier&
notifyObservers( const INotificationId& nId);
```

Inherited Protected Functions

INotifier		
addObserver	notifyObservers	observerList

Public Data

Notification Event Descriptions

These INotificationId strings are used for all notifications that an 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	

Chapter 88. IThread

Derivation

```
IBase
 IVBase
  IThread
```

Inherited By

```
ICurrentThread
```

Header File

```
ithread.hpp
```

Members

Member	Page
Constructor	537
adjustPriority	550
asDebugInfo	539
asString	540
autoInitGUI	540
current	548
currentHandle	549
currentId	549
defaultAutoInitGUI	540
defaultQueueSize	542
defaultStackSize	544
dialogControls	541
handle	549
id	549
isStarted	549
messageQueue	542
newStartedThread	552
operator =	552
priorityClass	550
priorityLevel	551
queueSize	543
relatedHandlesList	541
resume	546
setAutoInitGUI	540
setDefaultAutoInitGUI	541
setDefaultQueueSize	543
setDefaultStackSize	544
setPriority	551
setQueueSize	543
setRelatedHandlesList	541
setStackSize	545
setVariable	549
setWindowList	542
stackSize	545
start	546
startedThread	552
stop	548
stopProcessingMsgs	541
suspend	548
variable	550
windowList	542
~IThread	539

The IThread class represents threads of execution within the current program. You can use this class to implement multithreaded applications. Objects of this class give you access to all of the tasking APIs of the operating system. In addition, these objects serve as the anchor for thread-specific information.

Generally, you use objects of this class in one of the following ways:

- To apply thread functions to the current thread. In most cases, these functions are applied to the IThread object reference returned by the static member function `IThread::current` (p. 548).
- To create additional threads of execution by creating new objects of this class and starting them.
- To manipulate threads of execution initiated using alternate means, for which only the thread's identifier is known.

OS/390: To run in a multi-threaded environment, the kernel must be active on the system. In addition, the POSIX(ON) flag is required at runtime your application uses the IThread class or creates threads via `pthread_create`.

Presentation Manager: Compile multithreaded programs (those that call the `IThread::start` (p. 546) member function) with `/Gm+` to avoid unresolved externals at link time.

Windows (Win32s): Win32s does not support multiple threads per process. Applications running on this environment must be limited to a single thread.

Creating instances of this class or calling its member functions will not create and start threads on the system. Attempting to start a thread will result in an exception.

Motif: The AIX release of the Application Support Class Library does not support multithreading. The only access that an AIX application should need to IThread is exported through the class `ICurrentApplication` (p. 443) with the call to `ICurrentApplication::run` (p. 445). This call starts the processing of messages for the application.

You cannot control the priority of the AIX thread. The main thread of execution is a process. The only control that a user has over the priority of a process is using the nice values, which are not supported.

Portability Considerations: The AIX release of the Application Support Class Library does not presently support multiple threads per process. Portable applications must be limited to a single thread.

The OS/2 version of IThread has functions to control the size of the message queue and the stack size associated with a thread. These functions are not needed in the AIX environment and have no effect on the execution of a thread. You can call these functions in the AIX environment with no adverse side effects. Therefore, they have been left in the interface for ease of porting applications to multiple platforms.

Public Functions

Constructors

You can construct, copy, assign, and destruct objects of this class.

IThread

```
IThread();
```

Use this, the default constructor, to create an object of this class so that you can subsequently start using IThread::start (p. 546).

Windows (Win32s): Creating an instance of IThread with this constructor does not create or start a thread in the system.

```
IThread( const IThread& thread);
```

thread Reference to an object of this class.

Use this constructor if you would like to make a copy of an existing object of this class. This is commonly referred to as a copy constructor.

Windows (Win32s): Creating an instance of IThread with this constructor does not create or start a thread in the system.

```
IThread(
    const IThreadId& threadID,
    const IThreadHandle& threadHandle = IThreadHandle::noHandle);
```

threadID Reference to a thread identifier of a previously started thread.

threadHandle

Optional reference to a thread handle of a previously started thread.

Use this constructor to create an object of this class from the thread identifier (ID) of a previously started thread. You can use this constructor to provide this class' functionality to threads created using alternate means (for example, the native operating system's system calls).

Presentation Manager: The *threadHandle* parameter is ignored.

Windows: You must specify the *threadHandle* argument if the *threadId* argument does not refer to the current thread. You can use the return value of the CreateThread API for the thread handle.

Windows (Win32s): Creating an instance of IThread with this constructor does not create or start a thread in the system.

```
IThread( const IReference < IThreadFn >& threadFunction,
        Boolean autoInitGUI = defaultAutoInitGUI ( ));
```

threadFunction

Reference to an object of the IReference template class that was created using a derived IThreadFn class.

autoInitGUI

Boolean value that you use to specify if the thread is a GUI thread.

Use this constructor to create an object of this class when you need to specify a function to run that you have defined by deriving a new class from the IThreadFn (p. 559) class. This constructor passes the derived IThreadFn using an object of the IReference (p. 511) class so that the derived IThreadFn can be deleted (if necessary) when the thread terminates.

Use this form of the constructor to create a new object of this class and immediately dispatch it. This is equivalent to using the default constructor and then dispatching the thread using IThread::start (p. 546).

This constructor permits you to specify whether or not the new thread is to be a GUI thread. If it is, ICurrentThread::initializeGUI (p. 451) is called automatically after the thread is started.

Windows (Win32s): Creating an instance of IThread with this constructor throws an exception.

Exceptions	
InvalidRequest	The thread cannot be started; Win32s does not support multiple threads.

```
IThread( OptlinkFnPtr function,
         void* functionArgument,
         Boolean autoInitGUI = defaultAutoInitGUI ( ));
```

function Pointer to a function that uses the _Optlink calling convention.

functionArgument

Pointer to a parameter that you are passing to the function pointed to by *function*.

autoInitGUI

Boolean value that you use to specify that the thread is a GUI thread.

Use this constructor to create an object of this class when you need to specify a function to run that is defined with the _Optlink calling convention. Such functions are typically started using the function _beginThread.

Use this form of the constructor to create a new object of this class and immediately dispatch it. This is equivalent to using the default constructor and then dispatching the thread using IThread::start (p. 546).

This constructor permits you to specify whether the new thread is to be a GUI thread. If it is, ICurrentThread::initializeGUI (p. 451) is called automatically after the thread is started.

Windows (Win32s): Creating an instance of IThread with this constructor throws an exception.

Exceptions	
InvalidRequest	The thread cannot be started; Win32s does not support multiple threads.


```
IThread( SystemFnPtr function,
         unsigned long functionArgument,
         Boolean autoInitGUI = defaultAutoInitGUI ( ));
```

function Pointer to a function that uses the `_System` calling convention.

functionArgument

Unsigned long parameter that you are passing to the function pointed to by *function*.

autoInitGUI

Boolean value that you use to specify that the thread is a GUI thread.

Use this constructor to create an object of this class when you need to specify a function to run that you have defined using the `_System` calling convention. Such functions are typically started using the thread APIs that are defined for the operating system.

Use this form of the constructor to create a new object of this class and immediately dispatch it. This is equivalent to using the default constructor and then dispatching the thread using `IThread::start` (p. 546).

This constructor permits you to specify whether or not the new thread is to be a GUI thread. If it is, `ICurrentThread::initializeGUI` (p. 451) is called automatically after the thread is started.

Windows (Win32s): Creating an instance of `IThread` with this constructor throws an exception.

Exceptions	
<code>InvalidRequest</code>	The thread cannot be started; Win32s does not support multiple threads.

~IThread

Deallocates thread-related resources.

Note: When objects of this class are destructed, the thread is not terminated.

```
virtual
~IThread();
```

Windows (Win32s): This member function has no effect.

Diagnostics

Use these members for diagnostic purposes. They return an `IString` representation of an object of this class.

asDebugInfo

Provides debugging information about the class object. Use it to return general diagnostic information about the thread.

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

IThread

asString

Provides debugging information about the class object. Use it to return a string of the form "IThread(tid)", where *tid* represents the thread identifier.

```
virtual IString  
    asString() const;
```

Graphical User Interface (GUI) Support

Use these members to query and set the basic GUI support for threads.

autoInitGUI

Determines if this thread's GUI support is automatically initialized.

```
virtual Boolean  
    autoInitGUI() const;
```

OS/390: This member function is not supported.

Presentation Manager: This member obtains the setting of the flag that automatically initializes Presentation Manager for this thread.

defaultAutoInitGUI

Determines the default GUI support for automatic initialization. Unless the support is explicitly set using IThread::setDefaultAutoInitGUI (p. 541), this function returns true for threads running in a GUI session and false for threads running in a non-GUI session.

```
static Boolean  
    defaultAutoInitGUI();
```

OS/390: This member function is not supported.

Windows: Unless the support is explicitly set using IThread::setDefaultAutoInitGUI (p. 541), this function always returns true.

setAutoInitGUI

Sets the automatic initialization state for this thread. Set *initFlag* to true, the default, for GUI-based threads and to false for non-GUI-based threads.

```
virtual IThread&  
    setAutoInitGUI( Boolean initFlag = true);
```

initFlag Boolean value that determines if GUI support is automatically initialized for this thread.

OS/390: This member function is not supported.

Motif: Regardless of the value set by this function, the GUI is always initialized in AIX because of the Application Support Class Library's single-threaded limitation.

setDefaultAutoInitGUI

Sets the default GUI support for automatic initialization. Set *initFlag* to true, the default, for GUI-based threads and to false for non-GUI-based threads.

```
static void
    setDefaultAutoInitGUI( Boolean initFlag = true);
```

initFlag Boolean value that determines if GUI support is automatically initialized for this thread.

OS/390: This member function is not supported.

Motif: Regardless of the value set by this function, the GUI is always initialized in AIX because of the Application Support Class Library's single-threaded limitation.

stopProcessingMsgs

Terminates the processing of window events (messages) initiated by ICurrentThread::processMsgs (p. 452).

```
virtual IThread&
    stopProcessingMsgs();
```

OS/390: This member function is not supported.

Exceptions	
InvalidRequest	The event processing was not terminated. The termination failed because the event processing was not initiated by ICurrentThread::processMsgs.

Implementation

These members provide utilities used to implement this class. They are used by the Application Support Class Library and are not intended for use by users of this class.

dialogControls

Obtains a pointer to the dialog controls list for this thread.

```
IDialogControls*
    dialogControls() const;
```

OS/390: This member function is not supported.

relatedHandlesList

Obtains a pointer to the related handles list for this thread.

```
IRelatedHandlesList*
    relatedHandlesList() const;
```

setRelatedHandlesList

Sets a pointer to the related handles list for this thread.

```
IThread&
    setRelatedHandlesList( IRelatedHandlesList* list);
```

IThread

list Pointer to an IRelatedHandlesList object to set for this thread.

setWindowList

Sets a pointer to the window information list for this thread.

```
IThread&  
    setWindowList( IWindowList* list);
```

list Pointer to an IWindowList object to set for this thread.

OS/390: This member function is not supported.

windowList

Obtains a pointer to the window information list for this thread.

```
IWindowList*  
    windowList() const;
```

OS/390: This member function is not supported.

Message Queue

Use these members to query and set message queue information for graphical user interface (GUI) support.

defaultQueueSize

Obtains the default message queue size for the threads. The Application Support Class Library default queue size is 30.

Note: The Application Support Class Library uses this value by default for new objects of this class.

```
static long  
    defaultQueueSize();
```

OS/390: This member function is not supported.

Presentation Manager: This member obtains the default Presentation Manager message queue size to use for new IThreads.

Windows: This function returns the queue size that is set by IThread::setDefaultQueueSize (p. 543).

Motif: You cannot control the queue size on the X environment. This function returns the queue size that is set using IThread::setDefaultQueueSize (p. 543). Initially, this value is 0.

messageQueue

Returns the handle of the message queue. A 0 is returned if the handle does not exist or is invalid.

```
IMessageQueueHandle  
    messageQueue();
```

Use this version of the function if you want to reset an invalid message queue handle to 0. Do this if the handle is determined to be invalid.

```
IMessageQueueHandle  
    messageQueue() const;
```

Use this version of the function if you want the invalid message queue handle to remain invalid.

OS/390: This member function is not supported.

queueSize

Obtains the message queue size for the thread. This is the thread used by the application to receive events from the operating system.

```
virtual long  
    queueSize() const;
```

OS/390: This member function is not supported.

Windows: This function returns the queue size that is set by IThread::setQueueSize (p. 543).

Motif: You cannot control the size of the message queue in the X environment. This function returns the value set by IThread::setQueueSize (p. 543). The default value is returned by IThread::defaultQueueSize (p. 542).

setDefaultQueueSize

Sets the default message queue size for threads. The Application Support Class Library default queue size is 30.

Note: The Application Support Class Library uses this value by default for new objects of this class.

```
static void  
    setDefaultQueueSize( long queueSize);
```

aSize Long value that represents the default message queue size.

OS/390: This member function is not supported.

Windows: This function has no effect on the message queue size. The value set by this function is the value returned by IThread::defaultQueueSize (p. 542).

Motif: You cannot control the size of the queue in the X environment. This function has no effect on the message queue size. The value set by this function is the value returned by IThread::defaultQueueSize (p. 542).

setQueueSize

Sets the GUI message queue size for this thread.

```
virtual IThread&  
    setQueueSize( long queueSize);
```

queueSize
 Long value that represents the new message queue size.

OS/390: This member function is not supported.

Windows: This function has no effect on the size of the message queue. The value set by this function is the value returned by IThread::queueSize (p. 543).

Motif: You cannot control the message queue size for X. This function has no effect on the size of the message queue. The value set by this function is the value returned by IThread::queueSize (p. 543).

Stack Size

Use these members to query and set this thread's stack size.

defaultStackSize

Obtains the default stack size for threads in bytes.

Note: The Application Support Class Library uses this value by default for new objects of this class.

```
static unsigned long  
    defaultStackSize();
```

Windows (Win32s): You cannot control the stack size for a thread in this environment. This function returns the stack size that is set using IThread::setDefaultStackSize (p. 544). Initially, this value is 32768.

Motif: You cannot control the stack size for a thread in the X environment. This function returns the stack size that is set using IThread::setDefaultStackSize (p. 544). Initially, this value is 32768.

setDefaultStackSize

Sets the default stack size for threads in bytes.

Note: The Application Support Class Library uses this value by default for new objects of this class.

```
static void  
    setDefaultStackSize( unsigned long size);
```

size Unsigned long value that represents the default stack size in bytes.

OS/390: This member has no effect. System default stack size is used for all threads.

Windows (Win32s): You cannot control the size of the stack in this environment. This function has no effect on the size of the stack. The value set by this function is the value returned by IThread::defaultStackSize (p. 544).

Motif: You cannot control the size of the stack in the AIX environment. This function has no effect on the size of the stack. The value set by this function is the value returned by IThread::defaultStackSize (p. 544).

setStackSize

Sets this thread's stack size in bytes. If you have already started the thread, this value takes effect only when the thread is stopped and restarted.

```
virtual IThread&
    setStackSize( unsigned long size);
```

size Unsigned long value that represents the stack size in bytes.

OS/390: This member has no effect. System default stack size is used for all threads.

Windows: This value controls the amount of storage which will be committed to the thread by the system when the thread is started. Additional storage, up to the maximum available to your program, will be automatically allocated by the system if needed. You can control the maximum available stack available to your program by using the /STACKSIZE linker directive.

Windows (Win32s): You cannot control the stack size in this environment. This function has no effect on the size of the stack used. The value set by this function is the value returned by IThread::stackSize (p. 545).

Motif: You cannot control the stack size in AIX. This function has no effect on the size of the stack used. The value set by this function is the value returned by IThread::stackSize (p. 545).

stackSize

Obtains this thread's stack size in bytes.

```
virtual unsigned long
    stackSize() const;
```

Windows: Returns the value of the stack size that was set with the function IThread::setStackSize (p. 545). This value represents the stack size used as the initial stack size when the thread is created. For the primary thread, its value is the same as the initial default stack size. This value does not represent the actual committed stack in use by the thread.

You can control the maximum available stack available to your program by using the /STACKSIZE linker directive.

Windows (Win32s): You cannot control the stack size in this environment. This is not the actual size of the stack. This function returns the value of the stack size that was set with the function IThread::setStackSize (p. 545).

Motif: Returns the value of the stack size that was set with the function IThread::setStackSize (p. 545). You cannot control the size of the stack in AIX; therefore, this is not actually the size of the stack. The default value is returned by IThread::defaultStackSize (p. 544).

Starting and Stopping Threads

Use these members to start or stop threads.

resume

Resumes the thread's execution.

```
virtual void
  resume();
```

OS/390: This member function is not supported. An exception of type `invalidRequest` will result if this member is called.

Windows (Win32s): This member function has no effect.

Motif: The AIX release of the Application Support Class Library supports only a single-threaded environment. Therefore, you cannot suspend or resume a thread.

Exceptions	
<code>IAccessError</code>	The thread was not resumed. The thread was not suspended or the thread identifier (ID) is invalid.

start

Starts an asynchronous thread.

```
void
  start( const IReference < IThreadFn >& threadFunction,
        Boolean autoInitGUI = defaultAutoInitGUI ( ));
```

threadFunction

Reference to an object of the `IReference` template class that was created using a derived `IThreadFn` class.

autoInitGUI

Boolean value that you use to specify if the thread is a GUI thread.

You can start threads with a derived `IThreadFn` (p. 559) class. This function passes the derived `IThreadFn` using an object of the `IReference` (p. 511) class so that the derived `IThreadFn` can be deleted (if necessary) when the thread terminates.

This function permits you to specify whether the thread is started as a GUI thread. If it is, `ICurrentThread::initializeGUI` (p. 451) is called automatically after the thread is started.

Presentation Manager: The OS/2 version also allows threads to be started via the following means:

- With an `IThread::OptlinkFnPtr` type and `void*` parameter. This is to provide support for functions otherwise started using `_beginthread`.
- With an `IThread::SystemFnPtr` type and `unsigned long` parameter. This is to provide support for functions otherwise started using `DosCreateThread`.

Windows (Win32s): This member function has no effect.

Motif: Because the Application Support Class Library supports only single threading, there is no need to call this function.

```
void
    start( OptlinkFnPtr function,
           void* functionArgument,
           Boolean autoInitGUI = defaultAutoInitGUI ( ));
```

function Pointer to a function that uses the `_Optlink` calling convention.

functionArgument

Pointer to a parameter that you are passing to the function pointed to by *pfn*.

autoInitGUI

Boolean value that you use to specify if the thread is a GUI thread.

You can start threads with a function that is defined using the `_Optlink` calling convention. Such functions are typically started using the function, `_beginThread`. This function gives you the same capability.

This constructor permits you to specify whether the new thread is to be a GUI thread. If it is, `ICurrentThread::initializeGUI` (p. 451) is called automatically after the thread is started.

Windows (Win32s): This member function has no effect.

```
void
    start( SystemFnPtr function,
           unsigned long functionArgument,
           Boolean autoInitGUI = defaultAutoInitGUI ( ));
```

function Pointer to a function that uses the `_System` calling convention.

functionArgument

Unsigned long parameter that you are passing to the function pointed to by *pfn*.

autoInitGUI

Boolean value that you use to specify if the thread is a GUI thread.

You can start threads with a function that is defined using the `_System` calling convention. Such functions are typically started using the thread APIs that are defined for the operating system. This function gives you the same capability.

This constructor permits you to specify whether the new thread is to be a GUI thread. If it is, `ICurrentThread::initializeGUI` (p. 451) is called automatically after the thread is started.

Windows (Win32s): This member function has no effect.

stop

Stops the thread.

```
virtual void
    stop();
```

Presentation Manager: Do not use this function on suspended threads or the results can be unpredictable. This function is very powerful. Use it with extreme caution, preferably when the state of the target thread is known.

OS/390: A thread that is joined on a thread that is stopped has a status of -1 returned to it. Use this function with extreme caution, preferably when the state of the target thread is known. It may give an exception if the target thread is under a lock.

Windows (Win32s): This member function has no effect.

Exceptions	
IAccessError	The thread was not stopped. The thread was either busy or the thread identifier (ID) is invalid.

suspend

Suspends the thread's execution.

```
virtual void
    suspend();
```

OS/390: This member function is not supported. An exception of type invalidRequest will result if this member is called.

Windows (Win32s): This member function has no effect.

Motif: The Application Support Class Library supports only single threading; therefore, suspend and resume have no effect.

Exceptions	
IAccessError	The thread was not suspended. The thread identifier (ID) is invalid.

Thread Information

Use these members to query and set general thread information.

current

Returns a reference to an object of the ICurrentThread (p. 447) class that represents the currently executing thread.

```
static ICurrentThread&
    current();
```

currentHandle

Returns the thread handle for the current thread.

```
static IThreadHandle
    currentHandle();
```

Presentation Manager: The thread handle and thread ID are identical.

Windows: You must use the thread handle for the operating system thread and synchronization functions.

currentId

Obtains the identifier (ID) of the currently executing thread.

```
static IThreadId
    currentId();
```

Motif: The AIX release of the Application Support Class Library only supports a single thread. Therefore, the current thread ID is always 1.

handle

Returns the thread handle for the thread.

```
virtual IThreadHandle
    handle() const;
```

Presentation Manager: The thread handle and thread ID are identical.

Windows: You must use the thread handle for the operating system thread and synchronization functions.

id

Obtains the identifier (ID) of the thread. A return value of 0 indicates that the thread is not started.

```
virtual IThreadId
    id() const;
```

isStarted

Determines if this thread is currently active. If it is currently active, true is returned.

```
virtual Boolean
    isStarted() const;
```

setVariable

Stores data and an associated key, on a per thread instance. You can later retrieve this data using `IThread::variable` (p. 550). You can store up to 16 keys and their associated data per thread.

```
IThread&
    setVariable( const IString& key,
                 const IString& value);
```

key Reference to an `IString` object that represents the search key that is associated with the stored data.

value Reference to an `IString` object that contains the data to be stored.

Exceptions	
InvalidRequest	The keyed variable could not be set because the limit has been reached.

variable

Returns the data associated with the *key* that was stored using IThread::setVariable (p. 549).

```
IString
    variable( const IString& key) const;
```

key Reference to an IString object that represents the search key.

Thread Priority

Use these members to control the thread's priority. You can query or set the priority class and level.

adjustPriority

Changes the thread's priority level by the specified amount.

```
virtual IThread&
    adjustPriority( int delta);
```

delta Integer value that represents the priority level delta. This value must be between -31 and 31.

OS/390: This member has no effect.

Presentation Manager: The delta value is between -31 and 31.

Windows (Win32s): This member function has no effect.

Motif: You cannot adjust the thread's priority on AIX.

Exceptions	
IAccessError	The thread priority was not adjusted. The priority <i>delta</i> may be invalid. The valid values are -31 to 31.

priorityClass

Obtains the priority class of this thread. The return value is an enumerator provided by IApplication::PriorityClass (p. 425).

```
virtual IApplication::PriorityClass
    priorityClass() const;
```

Windows (Win32s): This member function has no effect.

Motif: The priority class for AIX is always IApplication::regular.

priorityLevel

Obtains the priority level of this thread. The return value is between 0 and 31.

```
virtual unsigned
    priorityLevel() const;
```

OS/390: This member has no effect.

Presentation Manager: The value of the level is in the range 0-31.

Windows (Win32s): This member function always returns 0.

Motif: The level for AIX is always 0.

setPriority

Sets the priority class and level of this thread.

```
virtual IThread&
    setPriority( IApplication::PriorityClass priority,
               unsigned level);
```

priority IApplication::PriorityClass (p. 425) enumerator that identifies the priority class.

level Unsigned value that represents the priority level.

OS/390: This member has no effect.

Windows (Win32s): This member function has no effect.

Motif: You cannot control the priority of a thread in the AIX environment. This function has no effect on the priority of an executing thread.

Exceptions	
IAccessError	The thread priority was not set. The priority class or level may be invalid.

Inherited Public Functions

IVBase		
asDebugInfo	asString	

IBase		
asDebugInfo	messageFile	setMessageFile
asString	messageText	version

Protected Functions

Constructors

You can construct, copy, assign, and destruct objects of this class.

operator =

Assigns the member data of an object of this class to another object of this class, therefore preserving resource allocation.

```
IThread&
    operator =( const IThread& thread);
```

thread Reference to an existing thread object.

Implementation

These members provide utilities used to implement this class. They are used by the Application Support Class Library and are not intended for use by users of this class.

newStartedThread

Creates an object of the IStartedThread class.

```
static IStartedThread*
    newStartedThread();
```

startedThread

Returns a pointer to the object of the IStartedThread class that corresponds to this thread.

```
virtual IStartedThread*
    startedThread() const;
```

Inherited Protected Data

IBase		
recoverable	unrecoverable	

Nested Classes

IThread contains the following nested classes:

IThread::Cursor (see page 555)

Nested Type Definitions

(void *)

```
typedef void ( _Optlink * OptlinkFnPtr ) ( void * );
```

This typedef defines a pointer to a function that uses the _Optlink calling convention.

Note: _Optlink is the default calling convention for IBM VisualAge for C/C++ on OS/2 and Windows. It has been included here for portability reasons.

(unsigned long)

```
typedef void ( _System * SystemFnPtr ) ( unsigned long );
```

This typedef defines a pointer to a function that uses the _System calling convention.

Note: _System is the default calling convention for the OS/2 operating system. It has been included here for portability reasons.

Chapter 89. IThread::Cursor

Derivation

IBase
IVBase
IThread::Cursor

Inherited By

None.

Header File

ithread.hpp

Members

Member	Page
Constructor	556
Cursor	556
invalidate	556
isValid	556
setToFirst	556
setToNext	556
threadId	556
~Cursor	556

The IThread::Cursor class creates and manages the cursor for an IThread (p. 535) object. This nested class defines objects that you can use to traverse or iterate through a set of active threads. In the same way that you can use a cursor to traverse the objects in a collection, you can use this cursor to traverse a set of threads one at a time.

Typically, you traverse a set, or collection, of active threads by doing the following:

1. Calling setToFirst (p. 556)
2. Looping through the threads using setToNext (p. 556)
3. Processing the returned thread IDs from threadId (p. 556) until isValid (p. 556) returns false

Note: This class only provides access to threads that an IThread (p. 535) object represents.

Public Functions

Constructors

Use these members to construct and destruct objects of this nested class.

Cursor

Constructs objects of this class. This constructor accepts an optional flag that indicates how threads are enumerated.

```
Cursor( Boolean allThreads = true);
```

allThreads

Boolean value that determines if all threads are enumerated or only those that have Application Support Class Library windows. The default is for all threads to be enumerated. Optional.

~Cursor

```
virtual  
~Cursor();
```

Thread Iteration

Use these members to iterate through the set of active threads.

invalidate

Marks the cursor as invalid.

```
virtual void  
invalidate();
```

isValid

Determines if the cursor is valid. If it is valid, true is returned.

```
Boolean  
isValid() const;
```

setToFirst

Sets the cursor's position to the first thread.

```
Boolean  
setToFirst();
```

setToNext

Advances the cursor's position to the next thread.

```
Boolean  
setToNext();
```

threadId

Obtains the identifier (ID) of the thread at the current cursor position.

```
IThreadId  
threadId() const;
```

Exceptions	
InvalidParameter	The thread identifier (ID) was not returned. The cursor is not valid.

Inherited Public Functions

IVBase		
asDebugInfo	asString	

IBase		
asDebugInfo	messageFile	setMessageFile
asString	messageText	version

Inherited Protected Data

IBase		
recoverable	unrecoverable	

Chapter 90. IThreadFn

Derivation

IBase
IVBase
 IRefCounted
 IThreadFn

Inherited By

IThreadMemberFn

Header File

ithread.hpp

Members

Member	Page
Constructor	559
run	560
~IThreadFn	560

The IThreadFn class represents functions to be dispatched on secondary threads of execution when you start an IThread (p. 535) object. The Application Support Class Library reference-counts objects of this class to manage their destruction after the thread has terminated. The Application Support Class Library passes such a reference to IThread::start (p. 546), supplying the reference via the IReference (p. 511) template class.

This class is an abstract thread function class.

Public Functions

Constructors

You cannot construct or destruct objects of this class because it is an abstract class. Use the template class IThreadMemberFn (p. 565) for dispatching C++ member functions to an object on a new thread.

IThreadFn

Provides the default and only constructor for this abstract class.

IThreadFn();

IThreadFn

~IThreadFn

```
virtual  
~IThreadFn();
```

Run Function

Use run function members to implement the code that you need to run on secondary threads of execution.

run

Called when the thread function object is called. Override this function to implement the code that you need to run on secondary threads of execution.

```
virtual void  
run() = 0;
```

Inherited Public Functions

IRefCounted		
addRef	removeRef	useCount

IVBase		
asDebugInfo	asString	

IBase		
asDebugInfo	messageFile	setMessageFile
asString	messageText	version

Inherited Protected Data

IBase		
recoverable	unrecoverable	

Chapter 91. IThreadHandle

Derivation

Inherits from none.

Inherited By

None.

Header File

ibhandle.hpp

Members

Member	Page
Constructor	561
asDebugInfo	562
asString	562
asUnsigned	562
noHandle	562

The IThreadHandle class is a wrapper for system-provided handles for threads.

Presentation Manager: An IThreadHandle is identical to an IThreadId. You can use them interchangeably.

Windows: Use IThreadHandle where a handle is needed, such as in the system thread and synchronization functions.

Public Functions

Constructors

You can construct objects of this class.

IThreadHandle

Constructs objects of this class from a thread handle (a value of the type IHandle::Value), which defaults to 0.

```
IThreadHandle( Value thread = 0);
```

Windows: If you are creating threads, use the return value from the CreateThread function to create a thread handle. You can also use IThread::currentHandle (p. 549) or ICurrentThread::handle (p. 448) to get the handle for the current thread.

Diagnostics

Use these members to get and set the accessible attributes of objects of this class.

IThreadHandle

asDebugInfo

Returns the handle as a string containing diagnostic information.

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

asString

Returns the handle as a string of form *nnn*.

```
IString  
    asString() const;
```

Note: For OS/390, the returned handle is not of form *nnn*. it is a string of hex numbers. .*

asUnsigned

Returns the handle value as an unsigned long value.

```
unsigned long  
    asUnsigned() const;
```

Public Data

Thread Handle Specifics

Contains thread-handle-specific values.

noHandle

Represents the default value in the IThread (p. 537) constructor for the *threadHandle* parameter. This parameter indicates that the thread is already accessible to the Application Support Class Library or is the current thread.

```
static const IThreadHandle  
    noHandle;
```

Nested Type Definitions

Value

```
typedef void * Value;
```


Chapter 92. IThreadId

Derivation

Inherits from none.

Inherited By

None.

Header File

ibhandle.hpp

Members

Member	Page
Constructor	563
asDebugInfo	564
asString	564
asUnsigned	564
isValid	564
operator =	564
operator pthread_t	564

The IThreadId class accesses numeric identifiers for threads.

Presentation Manager: IThreadId is an alias for the OS/2 Programmer's Toolkit type TID.

Motif: In AIX, the Application Support Class Library does not support multiple threads. This class always returns a ThreadId of 1.

Public Functions

Constructors

You can construct objects of this class.

IThreadId

Constructs objects of this class from a thread ID (a value of type IHandle::Value), which defaults to 0.

```
IThreadId( Value tid = 0);
```

Diagnostics

Use these members to get and set the accessible attributes of objects of this class.

IThreadId

asDebugInfo

Returns the handle as a string containing diagnostic information.

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

asString

Returns the handle as a string of form *nnn*.

```
IString  
    asString() const;
```

Note: For OS/390, the returned handle is not of form *nnn*. it is a string of hex numbers. .*

asUnsigned

Returns the handle value as an unsigned long value.

```
unsigned long  
    asUnsigned() const;
```

isValid

Returns true if the object is a valid thread ID.

```
IBase::Boolean  
    isValid() const;
```

Operators

This group contains operators for this class.

operator =

The assignment operator. You can assign one IThreadId to another or you can assign a pointer to the referenced type.

```
operator = (const IThreadId& tid);
```

operator pthread_t

Returns the POSIX thread ID.

```
operator pthread_t() const;
```

Chapter 93. IThreadMemberFn

Derivation

```
IBase
IVBase
  IRefCounted
    IThreadFn
      IThreadMemberFn
```

Inherited By

None.

Header File

ithread.hpp

Members

Member	Page
Constructor	566
run	566
~IThreadMemberFn	566

The IThreadMemberFn template class is derived from IThreadFn (p. 559) for dispatching C++ member functions to an object on a new thread. The template argument is the class of the object where the dispatched member functions are called.

The constructor for such objects requires the following:

- A reference to the object where the dispatched member functions are called
- A pointer to the member functions to call

The member functions return void and accept no parameters. You can add support for other member functions by deriving a class from this class.

Customization (Template Argument)

IThreadMemberFn is a template class that is created with the following template argument:

T The class of the object where the dispatched member functions are called.

Public Functions

Constructors

Use these members to construct and destruct objects of this template class.

IThreadMemberFn

IThreadMemberFn

You construct objects of this class by specifying an argument of the template argument class and a pointer to a member function of the template argument class.

```
IThreadMemberFn( T& obj,  
                 void ( T::* mem ) ( ));
```

obj Object of the template argument class **T**.
mem Pointer to a member function of the template argument class. The member function should return a *void* parameter and accept no formal parameters.

~IThreadMemberFn

```
virtual  
~IThreadMemberFn();
```

Run Function

Use these members to run the member function that was specified when an object of this class was created.

run

Calls the member function that you specify when you create an object of this class.

```
virtual void  
run();
```

Inherited Public Functions

IThreadFn		
run		

IRefCounted		
addRef	removeRef	useCount

IVBase		
asDebugInfo	asString	

IBase		
asDebugInfo	messageFile	setMessageFile
asString	messageText	version

Inherited Protected Data

IBase		
recoverable	unrecoverable	

Chapter 94. ITime Class

Derivation

```
IBase
ITime
```

Header File

```
itime.hpp
```

Objects of the ITime class represent units of time (hours, minutes, and seconds) as portions of days and provide support for converting these units of time into numeric and ASCII format. You can compare and operate on ITime objects by adding them to and subtracting them from other ITime objects. The ITime class returns locale-sensitive information, based on the current locale defined at runtime. See the description of the standard C function `setlocale` in the *OS/390 C/C++ Run-Time Library Reference* and *OS/390 C/C++ Programming Guide* for information about setting the locale.

A related class whose objects also represent units of time is the class `IDate` (see page 335).

Constructors

```
public:
ITime();
ITime(long seconds);
ITime(unsigned hours, unsigned minutes,
      unsigned seconds = 0);
ITime(const ITime &aTime);
```

Note: The following version of this constructor is available on OS/2 only.

```
public:
ITime(const _CTIME &cTime);
```

You can construct objects of this class in the following ways:

- By using the default constructor, which returns the current time
- By giving the number of seconds since midnight that the time is to represent

Note: If you specify a negative value for *seconds*, this constructor subtracts *seconds* from the number of seconds in a day
- By giving the number of hours, minutes, and seconds since midnight that the time is to represent

Note: You cannot specify a negative value for *seconds* when using this constructor
- By copying another ITime object
- By giving a container `CTIME` structure (OS/2 only)
- By using the static member function `ITime::now` (see page 570) to return the current time

Public Members

asCTIME

```
_CTIME asCTIME() const;
```

Returns the time as a container CTIME structure.

Note: This member function is available on OS/2 only.

asSeconds

```
long asSeconds() const;
```

Returns the number of seconds since midnight.

asString

```
IStrng asString(const char *fmt = " % X") const;
```

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 *OS/390 C/C++ Run-Time Library Reference* for more information about the `strftime` function.

The conversion specifiers that apply to ITime and their meanings are listed in the following table. `IDate::asString` (see page 336) 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 %.

hours

```
unsigned hours() const;
```

Returns the number of hours past midnight.

minutes

```
unsigned minutes() const;
```

Returns the number of minutes past the hour.

now

```
static ITime now();
```

Returns the current time.

Note: You can use this function as an ITime constructor.

operator!=

```
Boolean operator!=(const ITime &aTime) const;
```

Compares two objects to determine whether they are unequal.

operator+

```
ITime operator+(const ITime &aTime) const;
```

Adds two objects.

operator+=

```
ITime &operator+=(const ITime &aTime);
```

Adds two objects and stores the result in the receiver.

operator-

```
ITime operator-(const ITime &aTime) const;
```

Subtracts one object from another.

operator-=

```
ITime &operator-=(const ITime &aTime);
```

Subtracts one object from another and stores the result in the receiver.

operator<

```
Boolean operator<(const ITime &aTime) const;
```

Compares two objects to determine whether one is less than the other.

operator<<

```
friend ostream &operator<<(  
    ostream &aStream,  
    const ITime &aTime);
```

Outputs an object to an ostream.

operator<=

```
Boolean operator<=(const ITime &aTime) const;
```

Compares two objects to determine whether one is less than or equal to the other.

operator==

```
Boolean operator==(const ITime &aTime) const;
```

Compares two objects to determine whether they are equal.

operator>

```
Boolean operator>(const ITime &aTime) const;
```

Compares two objects to determine whether one is greater than the other.

ITime

operator>=

Boolean **operator>=**(const ITime &aTime) const;

Compares two objects to determine whether one is greater than or equal to the other.

seconds

unsigned **seconds**() const;

Returns the number of seconds past the minute.

Inherited Public Members

Member	Class	Page	Member	Class	Page
asDebugInfo	IBase	311	operator<<	IBase	312
asString	IBase	311			

Protected Members

initialize

ITime &**initialize**(long seconds);

A common initialization function used by the ITime constructors.

Chapter 95. ITimeStamp

Derivation

Inherits from none.

Inherited By

None.

Header File

itmstamp.hpp

Members

Member	Page
Constructor	575
asSeconds	577
asString	576
currentTimeStamp	575
operator !=	574
operator +	576
operator +=	576
operator -	576
operator -=	577
operator <	574
operator <=	574
operator ==	574
operator >	574
operator >=	574
operator IDate	577
operator ITime	577
secondsInDay	578

Objects of the ITimeStamp class represent a specific point in time. An ITimeStamp object can be created from an IDate object, an IDate and ITime object, or a value that represents the number of seconds from the reference date 01/01/2000 00:00:00. Use a negative value to represent a point in time before the reference date.

This class provides functions to obtain information about, compare, and manipulate ITimeStamp objects.

Public Functions

Comparisons

Use these members to compare two ITimeStamp objects.

ITimeStamp

operator !=

Returns true if the ITimeStamp objects are different points in time.

```
bool  
operator !=( const ITimeStamp& aTimeStamp) const;
```

operator <

Returns true if the left-hand ITimeStamp is earlier than the right-hand ITimeStamp.

```
bool  
operator <( const ITimeStamp& aTimeStamp) const;
```

operator <=

Returns true if the left-hand ITimeStamp is earlier than, or the same time as, the right-hand ITimeStamp.

```
bool  
operator <=( const ITimeStamp& aTimeStamp) const;
```

operator ==

Returns true if the ITimeStamp objects are the same point in time.

```
bool  
operator ==( const ITimeStamp& aTimeStamp) const;
```

operator >

Returns true if the left-hand ITimeStamp is later than the right-hand ITimeStamp.

```
bool  
operator >( const ITimeStamp& aTimeStamp) const;
```

operator >=

Returns true if the left-hand ITimeStamp is later than, or the same time as, the right-hand ITimeStamp.

```
bool  
operator >=( const ITimeStamp& aTimeStamp) const;
```

Constructors

You can create or copy objects of this class.

ITimeStamp

You can construct objects of this class by specifying a time in seconds, providing an IDate object, providing an IDate object and an ITime object, or copying an existing ITimeStamp object.

```
1 ITimeStamp( const ITimeStamp& aTimeStamp);
```

The copy constructor.

```
2 ITimeStamp( double seconds = 0.0);
```

Takes a value that represents the number of seconds from the reference date 01/01/2000 00:00:00. Specify a negative value to represent a time earlier than the reference date.

```
3 ITimeStamp( const IDate& aDate,
              const ITime& aTime);
```

Takes both an IDate and an ITime object, and creates an ITimeStamp object that represents the point in time they specify.

```
4 ITimeStamp( const IDate& aDate);
```

Takes an IDate object, and creates an ITimeStamp object that represents that point in time.

Current Date and Time

Use this member to obtain the current date and time on the user's system.

currentTimeStamp

Returns the current point in time (date and time) on the user's system, in the form of a new ITimeStamp object. This function can be used as an ITimeStamp constructor.

```
static ITimeStamp
    currentTimeStamp();
```

Diagnostics

Use this member to obtain an IString representation of an ITimeStamp object. You can use this member to write trace information when you are debugging.

asString

Returns the ITimeStamp as a string, which gives the time in date and time format. The format is system-dependent.

```
IString  
    asString() const;
```

Manipulations

Use these members to manipulate the value of an ITimeStamp object, or create a new ITimeStamp object based on the value of an existing one.

operator +

Adds the specified number of seconds to the ITimeStamp object, and returns the result as a new ITimeStamp object.

```
ITimeStamp  
    operator +( double seconds) const;
```

operator +=

Modifies the ITimeStamp object by adding the specified number of seconds.

```
ITimeStamp&  
    operator +=( double seconds);
```

operator -

```
1 double  
    operator -( const ITimeStamp& aTimeStamp) const;
```

Returns the difference in seconds between the two ITimeStamp objects. Returns a negative value if the left-hand ITimeStamp is earlier than the right-hand ITimeStamp.

```
2 ITimeStamp  
    operator -( double seconds) const;
```

Subtracts the specified number of seconds from the ITimeStamp object, and returns the result as a new ITimeStamp object.

operator -=

Modifies the ITimeStamp object by subtracting the specified number of seconds.

```
ITimeStamp&
operator -=( double seconds);
```

Queries

Use this member to obtain information about an ITimeStamp object.

asSeconds

Returns the time of the ITimeStamp object as the number of seconds from the reference date 01/01/2000 00:00:00. This number is negative if the time is earlier than the reference date.

```
double
asSeconds() const;
```

Type Conversions

Use these members to convert an ITimeStamp object to either an IDate object or an ITime object.

operator IDate

Returns an IDate object, which has the value of the ITimeStamp object, truncated to fit the IDate format: hour, minute, and second information is lost in the conversion. Day, month, and year information is preserved.

```
operator IDate() const;
```

operator ITime

Returns an ITime object, which has the value of the ITimeStamp object, truncated to fit the ITime format: day, month, and year information is lost in the conversion. Hour, minute, and second information is preserved, although the seconds are rounded down, and any partial seconds are lost.

```
operator ITime() const;
```

Public Data

ITimeStamp

Constants

secondsInDay

The number of seconds in a day (86400).

```
static const double  
    secondsInDay;
```


Chapter 96. ITrace Class

Derivation

```

IBase
  IVBase
    ITrace
  
```

Header File

```
itrace.hpp
```

Objects of the ITrace class provide module tracing within the Application Support 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 (see page 358)

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 Collection Class Library, the error ID contains the letters CLB, 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 Application Support 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()

- 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.

OS/390 C++-Specific Information: The default output location of tracing is standardOutput. Setting the output location of tracing to queue has the same effect as setting it to standardOutput. The __FUNCTION__ macro is not supported by the compiler; therefore the value defaults to the constant value f.

OS/2-Specific Information: 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.

Constructors

```
ITrace(const char *traceName = 0, long lineNumber = 0);
```

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.

traceName

(Optional) If you specify *traceName*, the name is written on construction and again on destruction.

Warning: If you pass an IString (see page 377) to the trace object, you must ensure that the lifetime of the IString exceeds the lifetime of the ITrace object. The library does not support the use of temporary IStrings.

lineNumber

(Optional) The line number where the trace statement occurred.

Public Members

disableTrace

```
static void disableTrace();
```

Disables the writing of trace entries.

disableWriteLineNumber

```
static void disableWriteLineNumber();
```

Disables the tracing of line number information.

disableWritePrefix

```
static void disableWritePrefix();
```

Disables the writing of the process ID, the thread ID, and the output line number to trace.

enableTrace

```
static void enableTrace();
```

Enables trace entries to be written.

enableWriteLineNumber

```
static void enableWriteLineNumber();
```

Enables the tracing of line number information.

enableWritePrefix

```
static void enableWritePrefix();
```

Enables the writing of the process ID, the thread ID, and the output line number to trace.

Portability Information

- On OS/390 C++, both processId and threadId are always set to 1.
- On UNIX implementations of VisualAge C++, threadId is always set to 1. processId is the id of the process.
- On AS/400 processId is always set to 0 and threadId is always set to 1.

isTraceEnabled

```
static Boolean isTraceEnabled();
```

Determines whether tracing is currently enabled.

isWriteLineNumberEnabled

```
static Boolean isWriteLineNumberEnabled();
```

Determines whether line numbers are currently being written.

isWritePrefixEnabled

```
static Boolean isWritePrefixEnabled();
```

Determines whether the line count prefix is being written.

traceDestination

```
static ITrace::Destination traceDestination();
```

Returns the trace output destination for this trace object. The returned value is an enumerator provided by ITrace::Destination (see page 583).

write

```
static void write(const IString &text);
static void write(const char *text);
```

Writes the specified text.

text The text to write as a character string.
text The text to write as an IString (see page 377).

writeToQueue

```
static void writeToQueue();
```

Sets the location for output to \\QUEUES\\PRINTF32.

Portability Information On UNIX and OS/390 operating systems, this member function is equivalent to writeToStandardOutput (see page 582).

writeToStandardError

```
static void writeToStandardError();
```

Sets the location for output to the standard error stream.

writeToStandardOutput

```
static void 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.

Inherited Public Members

Member	Class	Page	Member	Class	Page
asDebugInfo	IBase	311	asString	IVBase	314
asDebugInfo	IVBase	314	operator<<	IBase	312
asString	IBase	311	operator<<	IVBase	314

Protected Members

threadId

```
static unsigned long threadId();
```

Returns the current thread identifier.

Portability Information: In environments that do not support kernel threads, this function always returns a 1.

writeFormattedString

```
static void writeFormattedString(
    const IString &string, char *marker);
```

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 Application Support 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.

writeString

```
static void writeString(char *text);
```

Writes to the output device without formatting.

text Any trace information you want to write.

Enumerations

Destination

```
public:
enum 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.

Portability Information: When used on the following platforms, the queue enumerator is not supported, and queue tracing goes to stdout:

- AIX
- Solaris
- OS/390
- AS/400

Part 8. Appendixes, 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	587	parcel.h	596
circle.h	588	planet.h	598
curve.h	589	toyword.h	599
demoelem.h	591	transelm.h	600
dsur.h	592	trmapops.h	601
line.h	595	xebc2asc.h	602
graph.h	594		

These header files can be found in the PDS CBC.SCLBSAM.H. Some comments and white space have been removed.

animal.h

```

/*****
*
* Licensed Materials - Property of IBM
*
* 5645-001
* (C) Copyright IBM Corp. 1992, 1997
*
* US Government Users Restricted Rights - Use, duplication or
* disclosure restricted by GSA ADP Schedule Contract with IBM
* Corp.
*
*****/

// animal.h - Class Animal for use with the example animals.C

#include <globals.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;
    }
}

```

Example Header Files

```
friend ostream& operator<<(ostream& os, Animal const& p) {
    return os << "The " << p.name() << " is " << p.attribute()
        << "." << endl;
}

};

// Key access:
inline IString const& key(Ant 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(Ant const& animal, unsigned long n) {
    return hash(animal.name(), n);
}

#pragma checkout (resume)
#endif
```

circle.h

```
/*
 * Licensed Materials - Property of IBM
 *
 * 5645-001
 * (C) Copyright IBM Corp. 1992, 1997
 *
 * US Government Users Restricted Rights - Use, duplication or
 * disclosure restricted by GSA ADP Schedule Contract with IBM
 * Corp.
 */

#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;
    }
};

        #pragma checkout (resume)
        #endif

```

curve.h

```

/*****
*
*   Licensed Materials - Property of IBM
*
*   5645-001
*   (C) Copyright IBM Corp. 1992, 1997
*
*   US Government Users Restricted Rights - Use, duplication or
*   disclosure restricted by GSA ADP Schedule Contract with IBM
*   Corp.
*
*****/

#include <istring.hpp>

class Curve : public Graphics
{
public:

    float ivXStart;
    float ivYStart;
    float ivXFix1;
    float ivYFix1;
    float ivXFix2;
    float ivYFix2;
    float ivXFix3;
    float ivYFix3;
    float ivXEnd;
    float 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),

```

Example Header Files

```
        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()
    << endl
    << "with starting point: "
    << "(" << this->ivXStart << "|" <<
    << this->ivYStart << ")"
    << endl
    << "and with fix points: "
    << "(" << this->ivXFix1 << "|" << this->ivYFix1 << ")"
    << "(" << this->ivXFix2 << "|" << this->ivYFix2 << ")"
    << "(" << this->ivXFix3 << "|" << this->ivYFix3 << ")"
    << endl
    << "and with ending point: "
    << "(" << this->ivXEnd << "|" << this->ivYEnd << ")"
    << endl;
}

void          lengthOfCurve() const
{
    cout << "Length of "
    << Graphics::id()
    << " is: "
    << (sqrt(pow(((this->ivXFix1) - (this->ivXStart)),2)
            + pow(((this->ivYFix1) - (this->ivYStart)),2))
    + sqrt(pow(((this->ivXFix2) - (this->ivXFix1)),2)
            + pow(((this->ivYFix2) - (this->ivYFix1)),2))
    + sqrt(pow(((this->ivXFix3) - (this->ivXFix2)),2)
            + pow(((this->ivYFix3) - (this->ivYFix2)),2))
    + sqrt(pow(((this->ivXEnd) - (this->ivXFix3)),2)
            + pow(((this->ivYEnd) - (this->ivYFix3)),2)))
    << endl;
}
};

#pragma checkout (resume)
#endif
```

demoelem.h

```

/*****
 *
 * Licensed Materials - Property of IBM
 *
 * 5645-001
 * (C) Copyright IBM Corp. 1992, 1997
 *
 * US Government Users Restricted Rights - Use, duplication or
 * disclosure restricted by GSA ADP Schedule Contract with IBM
 * Corp.
 *
 *****/

/*-----\
 | demoelem.h - DemoElement for use with Key Collections
 |-----*/
#ifndef _DEMOELEM_H
#define _DEMOELEM_H

#include <stdlib.h>
#include <iglobals.h>
#include <iostream.h>
#include <istdops.h>

class DemoElement {

    int i;
    int 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;
    }

};

```

Example Header Files

```
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 implicitly created 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 undesirable,
// an element operations class must be used.

#endif

                                #pragma checkout (resume)
                                #endif
```

dsur.h

```
/*
 *
 * Licensed Materials - Property of IBM
 *
 * 5645-001
 * (C) Copyright IBM Corp. 1992, 1997
 *
 * US Government Users Restricted Rights - Use, duplication or
 * disclosure restricted by GSA ADP Schedule Contract with IBM
 * Corp.
 *
 */

/*-----*\
| dsur.h - Class for Disk Space Usage Records |
| This is used by the coding sample for the |
| Sorted Map and the Sorted Relation.       |
\*-----*/

#include <fstream.h>
#include <string.h>

#include <globals.h>

const int bufSize = 60;

class DiskSpaceUR {
    int    blocks;
    char*   name;

public:
    DiskSpaceUR() {}

    DiskSpaceUR (DiskSpaceUR const& dsur) {
        init(dsur);
    }

    void operator= (DiskSpaceUR const& dsur) {
        deInit();
        init(dsur);
    }
}
```

```

DiskSpaceUR (istream& DSURfile)      {
    DSURfile >> *this;
}

DiskSpaceUR () { deInit(); }

IBoolean operator == (DiskSpaceUR const& dsur) const {
    return (blocks == dsur.blocks)
        && strcmp (name, dsur.name) == 0;
}

friend istream& operator >> (istream& DSURfile,
                             DiskSpaceUR& dsur)      {
    DSURfile >> dsur.blocks;

    char temp[bufSize];
    DSURfile.get(temp, bufSize);

    if (DSURfile.good()) {
        // Remove leading tabs and blanks
        for (int cnt=0;
             (temp[cnt] == '\t') || (temp[cnt] == ' ');
             cnt++) {}
        dsur.name = new char[strlen(temp+cnt)+1];
        strcpy(dsur.name, temp+cnt);
    }
    else {
        dsur.setInvalid();
        dsur.name = new char[1];
        dsur.name[0] = '\0';
    }

    return DSURfile;
}

friend ostream& operator << (ostream& outstream,
                             DiskSpaceUR& dsur)      {
    outstream.width(bufSize);
    outstream.setf(ios::left, ios::adjustfield);
    outstream << dsur.name;

    outstream.width(9);
    outstream.setf(ios::right, ios::adjustfield);
    outstream << dsur.blocks;

    return outstream;
}

inline int const& space () const {return blocks;}

inline char* const& id () const {return name;}

inline IBoolean isValid () const {return (blocks > 0);}

protected:

inline void init (DiskSpaceUR const& dsur)      {
    blocks = dsur.blocks;
    name = new char[strlen(dsur.name) + 1];
    strcpy(name, dsur.name);
}

inline void deInit() { delete name; }

inline void setInvalid () { blocks = -1;}
};

// Key access on name
inline char* const& key (DiskSpaceUR const& dsur) {
    return dsur.id();
}

```

Example Header Files

```
    }

    // 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 can not use StdKeyOps here, because the 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(); }
};

                                #pragma checkout (resume)
                                #endif
```

graph.h

```
/******
*
* Licensed Materials - Property of IBM
*
* 5645-001
* (C) Copyright IBM Corp. 1992, 1997
*
* US Government Users Restricted Rights - Use, duplication or
* disclosure restricted by GSA ADP Schedule Contract with IBM
* Corp.
*
*****/

#include <istring.hpp>
#include <iostream.h>

class Graphics
{
protected:

    IString ivId;    /*** graphics ID ***/
    int     ivKey;    /*** graphics key ***/

public:

    Graphics (int graphicsKey, IString id) : ivKey(graphicsKey),
                                              ivId(id)
    { }

    Graphics()
    {
        cout << this->ivId
              << " will now be deleted ... "
              << endl;
    }

    IBoolean operator== (Graphics const& graphics) const
```



```

    {
        return (this->ivId == graphics.ivId);
    }

    IString const& id() const
    {
        return ivId;
    }

    virtual          void          draw() const =0;

    /**** This member function returns the graphic's key ****/
    /* Note that we are returning the int by reference, */
    /* because this member function will be used by the */
    /* key(...) function, which must return a reference. */
    /*****
    int const& graphicsKey() const
    {
        return ivKey;
    }

};

    /*****          key function          *****/
    /**** note that this interface must always be used with: ****/
    /****          Keytype const& key(...)          *****/
    /****          *****/
    /**** We are providing this key function for the element ****/
    /**** type Graphics and not for the managed pointer. ****/
    /*****
    inline int const& key (Graphics const& graphics)
    {
        return graphics.graphicsKey();
    }

                                #pragma checkout (resume)
                                #endif

```

line.h

```

    /*****
    *
    * Licensed Materials - Property of IBM
    *
    * 5645-001
    * (C) Copyright IBM Corp. 1992, 1997
    *
    * US Government Users Restricted Rights - Use, duplication or
    * disclosure restricted by GSA ADP Schedule Contract with IBM
    * Corp.
    *
    *****/

    #include <istring.hpp>
    #include <math.h>

    class Line : public Graphics
    {
    public:

        double ivXStart;
        double ivYStart;
        double ivXEnd;
        double ivYEnd;

        Line(int graphicsKey, IString id, double xstart, double ystart,
            double xend, double yend)

```

Example Header Files

```
        : 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;
}

};

        #pragma checkout (resume)
        #endif
```

parcel.h

```
/*
 *
 * Licensed Materials - Property of IBM
 *
 * 5645-001
 * (C) Copyright IBM Corp. 1992, 1997
 *
 * US Government Users Restricted Rights - Use, duplication or
 * disclosure restricted by GSA ADP Schedule Contract with IBM
 * Corp.
 *
 */
*****/

/*-----*\
| parcel.h - Class Parcel and its parts for use with the
|           example for Key Sorted Set and Heap.
|-----*/
#include <iostream.h>
```

```

// For definition of Boolean:
#include <iglobals.h>
// Class IString:
#include <istring.hpp>

class PlaceTime {
    IString cty;
    int daynum; // Keeping it simple: January 9 is day 9
public:
    PlaceTime(IString acity, int aday) : cty(acity), daynum(aday) {}
    PlaceTime(IString acity) : cty(acity) {daynum = 0;}
    IString const& city() const {
        return cty;
    }
    int const& day() const {
        return daynum;
    }
    void operator=(PlaceTime const& pt) {
        cty = pt.cty;
        daynum = pt.daynum;
    }
    IBoolean operator==(PlaceTime const& pt) const {
        return ( cty == pt.cty
            && (daynum == pt.daynum) );
    }
};

class Parcel {
    PlaceTime org, lstAr;
    IString dst, id;
public:
    Parcel(IString orig, IString dest, int day, IString ident)
        : org(orig, day), lstAr(orig, day), dst(dest), id(ident) {}
    void arrivedAt(IString const& acity, int const& day) {
        PlaceTime nowAt(acity, day);
        // Only if not already there before
        if (nowAt.city() != lstAr.city())
            lstAr = nowAt;
    }
    void wasDelivered(int const& day) {arrivedAt(dst, day); }
    PlaceTime const& origin() const {
        return org;
    }
    IString const& destination() const {
        return dst;
    }
    PlaceTime const& lastArrival() const {
        return lstAr;
    }
    IString const& ID() const {
        return id;
    }
}

```

Example Header Files

```
friend ostream& operator<<(ostream& os, Parcel const& p) {
    os << p.id << ": From " << p.org.city()
    << "(day " << p.org.day() << ") to " << p.dst;

    if (p.lstAr.city() != p.dst) {
        os << endl << "            is at " << p.lstAr.city()
        << " since day " << p.lstAr.day() << ".";
    }
    else {
        os << endl << "            was delivered on day "
        << p.lstAr.day() << ".";
    }
    return os;
}
};

// Key access:
inline IString const& key( Parcel const& p) {
    return p.ID();
}

// We need a compare function for the key.
// Let's use the default provided for IString:
inline long compare(Parcel const& p1, Parcel const& p2) {
    return compare(p1.ID(), p2.ID());
}

#pragma checkout (resume)
#endif
```

planet.h

```
/******
 *
 * Licensed Materials - Property of IBM
 *
 * 5645-001
 * (C) Copyright IBM Corp. 1992, 1997
 *
 * US Government Users Restricted Rights - Use, duplication or
 * disclosure restricted by GSA ADP Schedule Contract with IBM
 * Corp.
 *
 *****/

/*-----\
| planet.h - Class Planet for use in our Sorted Set example |
\*-----*/
#include <iostream.h>
#include <iappl.h>

class Planet {
private:
    char* pname;
    float dist;
    float mass;
    float bright;

public:
    // Let's 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); }
};

/*-----*\
|   Applicator   |
\*-----*/

class SayPlanetName : public IApplicator<Planet>  {
public:
    virtual IBoolean applyTo(Planet& p)
        { cout << " " << p.name() << " "; return True;}
};

                                #pragma checkout (resume)
                                #endif

```

toyword.h

```

/*****
*
*   Licensed Materials - Property of IBM
*
*   5645-001
*   (C) Copyright IBM Corp. 1992, 1997
*
*   US Government Users Restricted Rights - Use, duplication or
*   disclosure restricted by GSA ADP Schedule Contract with IBM
*   Corp.
*
*****/

/*-----*\
| toyword.h - Class Word for use with coding examples. |
\*-----*/

#include <istring.hpp>

class Word {

    IString      ivWord;
    unsigned int ivKey;

public:

    //Constructor to be used for sample: wordbag.c
    Word (IString const& word, unsigned theLength)
    : ivWord (word), ivKey (theLength) {}

    //Constructor to be used for sample: wordseq.c
    Word (IString const& word)
    : ivWord (word) {}

    IBoolean operator> (Word const& w1)
    { return ivWord > w1.ivWord;
    }

    unsigned int setKey ()
    { return (ivKey = ivWord.length());
    }

    IString const& getWord() const
    { return ivWord;
    }
}

```

Example Header Files

```
        unsigned int const& getKey() const
        { return ivKey;
        }
};

// Key access. The length of the word is the key.
inline unsigned int const& key (Word const& aWord)
{ return aWord.getKey();
}

        #pragma checkout (resume)
        #endif
```

transelm.h

```
/*
 * Licensed Materials - Property of IBM
 *
 * 5645-001
 * (C) Copyright IBM Corp. 1992, 1997
 *
 * US Government Users Restricted Rights - Use, duplication or
 * disclosure restricted by GSA ADP Schedule Contract with IBM
 * Corp.
 */
*****/

/*-----*\
| transelm.h - Class TranslationElement for use with the
| Translation Table example.
| \*-----*/
#ifndef _TRANSELM_H
#define _TRANSELM_H

#include <globals.h>

class TranslationElement {

    char ivAscCode;
    char ivEbcCode;

public:

    /* Let the compiler generate Default and Copy Constructor,*/
    /* Destructor and Assignment for us. */

    char const& ascCode () const
    { return ivAscCode;
    }

    char const& ebcCode () const
    { return ivEbcCode;
    }

    TranslationElement (char asc, char ebc)
        : ivAscCode(asc), ivEbcCode(ebc) {};

    /* We need to define the equality. */
    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

                                #pragma checkout (resume)
                                #endif

```

trmapops.h

```

/*****
 *
 *   Licensed Materials - Property of IBM
 *
 *   5645-001
 *   (C) Copyright IBM Corp. 1992, 1997
 *
 *   US Government Users Restricted Rights - Use, duplication or
 *   disclosure restricted by GSA ADP Schedule Contract with IBM
 *   Corp.
 *
 *****/

/*-----\
 | trmapops.h - Translation Map Operations
 | This is the base class for the element
 | operations for our 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

                                #pragma checkout (resume)
                                #endif

```

xebc2asc.h

```

/*****
 *
 * Licensed Materials - Property of IBM
 *
 * 5645-001
 * (C) Copyright IBM Corp. 1992, 1997
 *
 * US Government Users Restricted Rights - Use, duplication or
 * disclosure restricted by GSA ADP Schedule Contract with IBM
 * Corp.
 *
 *****/

/*-----*/
xebc2asc.h : EBCDIC - ASCII Translation Table.
Example: To convert from an EBCDIC blank, 40,
         use translationtable[40], which gives 20,
         the ASCII blank.
/*-----*/

const unsigned char translationTable[256] = {
/* 00 */ 0x00, 0x01, 0x02, 0x03, 0xCF, 0x09, 0xD3, 0x7F,
/* 08 */ 0xD4, 0xD5, 0xC3, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F,
/* 10 */ 0x10, 0x11, 0x12, 0x13, 0xC7, 0xB4, 0x08, 0xC9,
/* 18 */ 0x18, 0x19, 0xCC, 0xCD, 0x83, 0x1D, 0xD2, 0x1F,
/* 20 */ 0x81, 0x82, 0x1C, 0x84, 0x86, 0x0A, 0x17, 0x1B,
/* 28 */ 0x89, 0x91, 0x92, 0x95, 0xA2, 0x05, 0x06, 0x07,
/* 30 */ 0xE0, 0xEE, 0x16, 0xE5, 0xD0, 0x1E, 0xEA, 0x04,
/* 38 */ 0x8A, 0xF6, 0xC6, 0xC2, 0x14, 0x15, 0xC1, 0x1A,
/* 40 */ 0x20, 0xA6, 0xE1, 0x80, 0xEB, 0x90, 0x9F, 0xE2,
/* 48 */ 0xAB, 0x8B, 0x9B, 0x2E, 0x3C, 0x28, 0x2B, 0x7C,
/* 50 */ 0x26, 0xA9, 0xAA, 0x9C, 0xDB, 0xA5, 0x99, 0xE3,
/* 58 */ 0xA8, 0x9E, 0x21, 0x24, 0x2A, 0x29, 0x3B, 0x5E,
/* 60 */ 0x2D, 0x2F, 0xDF, 0xDC, 0x9A, 0xDD, 0xDE, 0x98,
/* 68 */ 0x9D, 0xAC, 0xBA, 0x2C, 0x25, 0x5F, 0x3E, 0x3F,
/* 70 */ 0xD7, 0x88, 0x94, 0xB0, 0xB1, 0xB2, 0xFC, 0xD6,
/* 78 */ 0xFB, 0x60, 0x3A, 0x23, 0x40, 0x27, 0x3D, 0x22,
/* 80 */ 0xF8, 0x61, 0x62, 0x63, 0x64, 0x65, 0x66, 0x67,
/* 88 */ 0x68, 0x69, 0x96, 0xA4, 0xF3, 0xAF, 0xAE, 0xC5,
/* 90 */ 0x8C, 0x6A, 0x6B, 0x6C, 0x6D, 0x6E, 0x6F, 0x70,
/* 98 */ 0x71, 0x72, 0x97, 0x87, 0xCE, 0x93, 0xF1, 0xFE,
/* A0 */ 0xC8, 0x7E, 0x73, 0x74, 0x75, 0x76, 0x77, 0x78,
/* A8 */ 0x79, 0x7A, 0xEF, 0xC0, 0xDA, 0x5B, 0xF2, 0xF9,
/* B0 */ 0xB5, 0xB6, 0xFD, 0xB7, 0xB8, 0xB9, 0xE6, 0xBB,
/* B8 */ 0xBC, 0xBD, 0x8D, 0xD9, 0xBF, 0x5D, 0xD8, 0xC4,
/* C0 */ 0x7B, 0x41, 0x42, 0x43, 0x44, 0x45, 0x46, 0x47,
/* C8 */ 0x48, 0x49, 0xCB, 0xCA, 0xBE, 0xE8, 0xEC, 0xED,
/* D0 */ 0x7D, 0x4A, 0x4B, 0x4C, 0x4D, 0x4E, 0x4F, 0x50,
/* D8 */ 0x51, 0x52, 0xA1, 0xAD, 0xF5, 0xF4, 0xA3, 0x8F,
/* E0 */ 0x5C, 0xE7, 0x53, 0x54, 0x55, 0x56, 0x57, 0x58,
/* E8 */ 0x59, 0x5A, 0xA0, 0x85, 0x8E, 0xE9, 0xE4, 0xD1,
/* F0 */ 0x30, 0x31, 0x32, 0x33, 0x34, 0x35, 0x36, 0x37,
/* F8 */ 0x38, 0x39, 0xB3, 0xF7, 0xF0, 0xFA, 0xA7, 0xFF
};

#pragma checkout (resume)
#endif

```


Appendix B. OS/390 C/C++ Class Library Runtime Messages

Messages for I/O Stream and Complex Mathematics Class Libraries

9900 An attempt to allocate memory has failed.

Explanation: The attempt at obtaining memory in order to satisfy the current library request has failed.

Programmer Response: Run the program in a larger region or use the HEAP(„FREE) run-time option instead of the HEAP(„KEEP) option.

System Action: The requested function will fail.

9901 IOStreams do not support Record Mode I/O.

Explanation: The application is attempting to initialize an IOStreams object to perform Record Mode I/O. Record Mode I/O is not supported in IOStreams objects.

Programmer Response: Remove the “type=record” specification from the constructor or open() function call.

System Action: The attempt to initialize the object has failed. Execution continues.

9902 too many characters

Explanation: The application called the form() function with a format specifier string that caused form() to write past the end of the format buffer. form() is an obsolete interface provided in stream.h for compatibility with old code.

Programmer Response: Split the call to form() into two or more calls.

System Action: Execution is aborted.

9903 singularity: log((0,0))

Explanation: The application is attempting to take the log of (0.0, 0.0).

Programmer Response: Correct the value passed to log() and resubmit.

System Action: Execution is aborted.

Messages for Application Support Class Library

9000 string overflow

Explanation: String overflow exception raised

Programmer Response: Ensure you have allocated enough buffer to hold string

System Action: None.

9001 string index error

Explanation: String index error exception raised

Programmer Response: Ensure your indexes are within range

System Action: None.

9002 Invalid DBCS String.

Explanation: DBCS characters in the MBCS string are not enclosed in shift-out and shift-in characters. Either shift-out or shift-in character is missing.

Programmer Response: Ensure DBCS characters within MBCS string are enclosed in shift-out and shift-in characters.

System Action: None.

9003 Error while converting MBCS string to Wide Char string.

Explanation: Most likely reason for this error is that the MBCS string is invalid. DBCS characters in the string are not enclosed in shift-out and shift-in characters. Either shift-out or shift-in character is missing.

Programmer Response: Ensure DBCS characters within MBCS string are enclosed in shift-out and shift-in characters.

System Action: None.

9004 Protected Function of class called, it can result in . unpredictable behavior.

Explanation: User application has called protected function of a class. This can result in unpredictable behavior.

Programmer Response: Change your application to ensure the protected function of the class is not called.

System Action: None.

9005 Unable to acquire a semaphore to satisfy the lock() request.

Explanation: There is no more semaphore resource available to complete the user request. Most likely the system limit for the number of semaphores has been exceeded.

Programmer Response: Free up unused semaphore resources that your application might have acquired and try the request again. If problem persists, contact your system representative to free the unused semaphore resources.

System Action: Check semaphore usage. If all semaphores are exhausted, then cancel some applications to free up the semaphores. If problem still persists, contact IBM support representative.

9006 Decimal data overflow.

Explanation: Target operand is too small to store the value of the operation.

Programmer Response: Change the size of the target operand.

System Action: None.

9007 The specified thread ID is not valid.

Explanation: User application has passed an invalid thread id to IThread class.

Programmer Response: Ensure a valid thread Id (pthread_t) embedded in threadID_t struct is passed to IThread Class constructor.

System Action: None.

9008 start() is not valid because the specified thread is already started

Explanation: User application has called start() function on IThread class but the thread is already running.

Programmer Response: Check your application to ensure that start() function is called after the previous function dispatched on the IThread has been completed.

System Action: None.

9009 Keyed variable could not be set because the limit has been exceeded.

Explanation: An attempt was made to allocate a keyed thread variable beyond the library's limit. This limit is 16.

Programmer Response: Check your application to ensure that the number of keyed thread variables are below the maximum limit.

System Action: None.

9010 Unsupported member function of IThread class called.

Explanation: User application has called a member function of IThread class which is not supported on this platform.

Programmer Response: Change your application logic to avoid calling this member function.

System Action: None.

9011 Class or the called member function is not supported.

Explanation: User application has called a member function of Class or has tried to instantiate an instance of a class which is supported only in MVS OE Environment.

Programmer Response: Change your application logic to avoid calling the member function or creating an instance of class which is not supported in MVS non-OE Environment.

System Action: None.

9050 The following Expression must be true, but evaluated to false: %1

Explanation: The expression must be true but it evaluated to false.

Programmer Response: Check the variables in the expression.

System Action: None.

9051 GUI Exception condition detected.

Explanation: GUI Exception condition detected

Programmer Response: None.

System Action: None.

9052 System Exception condition detected.

Explanation: System Exception condition detected

Programmer Response: None.

System Action: None.

Messages for Collection Class Library

9500 A child already exists.

Explanation: A child already exists at the given position.

Programmer Response: Check whether there is no child at the position you want to add one.

System Action: None, due to unfulfilled precondition.

9501 The collection is empty.

Explanation: The collection is empty.

Programmer Response: Check your program to ensure that you added at least one element to the collection.

System Action: None, due to unfulfilled precondition.

9502 The cursor is not contained in collection.

Explanation: The cursor is not contained in collection, the corresponding element might have been removed from the collection.

Programmer Response: Check your program to ensure that the cursor points to an element of the collection.

System Action: None, due to unfulfilled precondition.

9503 The cursor is not for given collection.

Explanation: The cursor does not belong to the given collection

Programmer Response: Check your program to ensure that the cursor points to an element belonging to the given collection.

System Action: None, due to unfulfilled precondition.

9504 The cursor is not for this collection.

Explanation: The cursor does not belong to the collection to which the collection member function - like `setToNext` - issuing this message is applied.

Programmer Response: Check your program to ensure that the cursor you specify with the collection member function is valid for the collection that function is applied to.

System Action: None, due to unfulfilled precondition.

9505 An identical collection was specified.

Explanation: Occurs when the function `addAllFrom` is called with the source collection being the same as the target collection.

Programmer Response: Check your program to ensure that the collections are different.

System Action: None, due to unfulfilled precondition.

9506 an invalid cursor was specified.

Explanation: The cursor points to an invalid position that means at that position there is not an object which could be an element of the collection.

Programmer Response: Check your program to ensure that the cursor points to a valid position.

System Action: None, due to unfulfilled precondition.

9507 An invalid position was specified.

Explanation: The position specified with a function applied to a collection is invalid for this collection.

Programmer Response: Check your program to ensure that the position is valid for the collection you want to apply the function.

System Action: None, due to unfulfilled precondition.

9508 An invalid replacement was specified.

Explanation: Occurs when, during a `replaceAt` function, the replacing element has different positioning properties than the positioning properties of the element to be replaced.

Programmer Response: Check your program to ensure that the replacing elements has the same positioning properties as the element the cursor points to.

System Action: None, due to unfulfilled precondition.

9509 A key already exists.

Explanation: Occurs when a function attempts to add an element to a map or sorted map that already has a different element with the same key.

Programmer Response: Check your program to ensure that the key of the element to be added is different from all keys of the elements of the map.

System Action: None, due to unfulfilled precondition.

9510 A key is not contained.

Explanation: Occurs when the function `elementWithKey` is applied to a collection that does not contain an element with the specified key.

Programmer Response: Check your program to ensure that the collection contains an element with the given key.

System Action: None, due to unfulfilled precondition.

9511 This collection is unbounded.

Explanation: Occurs if the function `maxNumberOfElements` is applied to a collection that is not bounded

Programmer Response: Check your program to ensure that the collection is bounded or do not apply the function `maxNumberOfElements` to it.

System Action: None, due to unfulfilled precondition.

9512 The system is out of memory for collection elements.

Explanation: Occurs when a function cannot obtain the space that is requires.

Programmer Response: Check that the system resources offer enough memory.

System Action: None.

9513 A root already exists.

Explanation: Occurs when the function `addAsRoot` is called for a tree that already has a root.

Programmer Response: Check your program to ensure that the root does not yet exist in your tree.

System Action: None, due to unfulfilled precondition.

9514 A cyclic child attachment occurred.

Explanation: Occurs when you try to attach a subtree to one of its own children.

Programmer Response: Check your program to ensure that you do not try to attach a subtree to one of its own children.

System Action: None, due to unfulfilled precondition.

9515 Internal mutex error occurred.

Explanation: Occurs when you try to create a Guard and there are no more mutexes available.

Programmer Response: Check the OS environment parameters. If possible increase the number of possible concurrent threads/mutexes.

System Action: None.

9516 Internal lock error occurred.

Explanation: An error occurred during an internal lock call.

Programmer Response: Check the system environment and reduce the number of threads if possible. Rerun the application.

System Action: None

9517 A timeout occurred.

Explanation: A Guard was requested with a specified time-out value. The internal lock request was not successful.

Programmer Response: Check your application locking sequence, check if all Guard destructors are called, try to increase the time-out value.

System Action: None

9518 Internal unlock error occurred.

Explanation: An error occurred during an internal unlock call. The internal lock request was not successful.

Programmer Response: Check the system environment and reduce the number of threads if possible. Rerun the application.

System Action: None

9900 An attempt to allocate memory has failed.

Explanation: The attempt at obtaining memory in order to satisfy the current library request has failed.

Programmer Response: Run the program in a larger region or use the `HEAP(,FREE)` run-time option instead of the `HEAP(,KEEP)` option.

System Action: The requested function will fail.

9901 IOStreams do not support Record Mode I/O.

Explanation: The application is attempting to initialize an IOStreams object to perform Record Mode I/O. Record Mode I/O is not supported in IOStreams objects.

Programmer Response: Remove the "type=record" specification from the constructor or `open()` function call.

System Action: The attempt to initialize the object has failed. Execution continues.

9902 Too many characters.

Explanation: The application called the `form()` function with a format specifier string that caused `form()` to write past the end of the format buffer. `form()` is an obsolete interface provided in `stream.h` for compatibility with old code.

Programmer Response: Split the call to `form()` into two or more calls.

System Action: Execution is aborted.

9903 Singularity: log((0,0)).

Explanation: The application is attempting to take the log of (0.0, 0.0).

Programmer Response: Correct the value passed to log() and resubmit.

System Action: Execution is aborted.

9904 Internal error: pthread_mutex_destroy() failed.

Explanation: The attempt to release the mutex handle failed.

Programmer Response: Note return code and errno to identify the cause of the problem and inform the IBM support.

System Action: Execution is aborted.

9905 Internal error: pthread_mutex_lock() failed.

Explanation: The attempt to lock the mutex handle failed.

Programmer Response: Note return code and errno to identify the cause of the problem and inform the IBM support.

System Action: Execution is aborted.

9906 Internal error: pthread_mutex_unlock() failed.

Explanation: The attempt to unlock the mutex handle failed.

Programmer Response: Note return code and errno to identify the cause of the problem and inform the IBM support.

System Action: Execution is aborted.

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*, ANSI/ISO X3.172-1990, copyright 1990 by the American National Standards Institute (ANSI/ISO). Copies may be purchased from the American National Standards Institute, 1430 Broadway, New York, New York 10018. Such definitions are indicated by the symbol *ANSI/ISO* 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 have a direct object of an abstract class. See also *base class*. (2) A class that allows polymorphism. There can be no objects of an abstract class; they are only used to derive new classes.

abstract code unit. See *ACU*.

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). A data type with a private representation and a public set of operations (functions or operators) which restrict access to that data type to that set of operations. The C++ language uses the concept of classes to implement data abstraction.

access. An attribute that determines whether or not a class member is accessible in an expression or declaration.

access declaration. A declaration used to restore access to members of a base class.

access mode. (1) A technique that is used to obtain a particular logical record from, or to place a particular logical record into, a file assigned to a mass storage device. *ANSI/ISO*. (2) The manner in which files are referred to by a computer. Access can be sequential (records are referred to one after another in the order in which they appear on the file), access can be random (the individual records can be referred to in a nonsequential manner), or access can be dynamic (records can be accessed sequentially or randomly, depending on the form of the input/output request). *IBM*. (3) A particular form of access permitted to a file. *X/Open*.

access resolution. The process by which the accessibility of a particular class member is determined.

access specifier. One of the C++ keywords: public, private, and protected, used to define the access to a member.

ACU (abstract code unit). A measurement used by the OS/390 C/C++ compiler for judging the size of a function. The number of ACUs that comprise a function is proportional to its size and complexity.

addressing mode. See *AMODE*.

address space. (1) The range of addresses available to a computer program. *ANSI/ISO*. (2) The complete range of addresses that are available to a programmer. See also *virtual address space*. (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*.

alert. (1) A message sent to a management services focal point in a network to identify a problem or an impending problem. *IBM.* (2) To cause the user's terminal to give some audible or visual indication that an error or some other event has occurred. When the standard output is directed to a terminal device, the method for alerting the terminal user is unspecified. When the standard output is not directed to a terminal device, the alert is accomplished by writing the alert character to standard output (unless the utility description indicates that the use of standard output produces undefined results in this case). *X/Open.*

alert character. A character that in the output stream should cause a terminal to alert its user via a visual or audible notification. The alert character is the character designated by a '\a' in the C and C++ languages. It is unspecified whether this character is the exact sequence transmitted to an output device by the system to accomplish the alert function. *X/Open.*

This character is named <alert> in the portable character set.

alias. (1) An alternate label; for example, a label and one or more aliases may be used to refer to the same data element or point in a computer program.

ANSI/ISO. (2) An alternate name for a member of a partitioned data set. *IBM.* (3) An alternate name used for a network. Synonymous with nickname. *IBM.*

alias name. (1) A word consisting solely of underscores, digits, and alphabets from the portable file name character set, and any of the following characters: ! % , @. Implementations may allow other characters within alias names as an extension. *X/Open.* (2) An alternate name. *IBM.* (3) A name that is defined in one network to represent a logical unit name in another interconnected network. The alias name does not have to be the same as the real name; if these names are not the same; translation is required. *IBM.*

alignment. The storing of data in relation to certain machine-dependent boundaries. *IBM.*

alternate code point. A syntactic code point that permits a substitute code point to be used. For example, the left brace ({) can be represented by X'B0' and also by X'C0'.

American National Standard Code for Information Interchange (ASCII). 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. *IBM.*

Note: IBM has defined an extension to ASCII code (characters 128–255).

American National Standards Institute (ANSI/ISO).

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. *ANSI/ISO.*

AMODE (addressing mode). In MVS, a program attribute that refers to the address length that a program is prepared to handle upon entry. In MVS, addresses may be 24 or 31 bits in length. *IBM.*

angle brackets. The characters < (left angle bracket) and > (right angle bracket). When used in the phrase "enclosed in angle 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 <less-than-sign> and <greater-than-sign> are used. *X/Open.*

anonymous union. A union that is declared within a structure or class and does not have a name. It must not be followed by a declarator.

ANSI/ISO. See *American National Standards Institute.*

API (application program interface). A functional interface supplied by the operating system or by a separately orderable licensed program that allows an application program written in a high-level language to use specific data or functions of the operating system or the licensed program. *IBM.*

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.*

application generator. An application development tool that creates applications, application components (panels, data, databases, logic, interfaces to system services), or complete application systems from design specifications.

application program. A program written for or by a user that applies to the user's work, such as a program that does inventory control or payroll. *IBM.*

archive libraries. The archive library file, when created for application program object files, has a special symbol table for members that are object files.

argument. (1) A parameter passed between a calling program and a called program. *IBM.* (2) In a function call, an expression that represents a value that the calling function passes to the function specified in the call. Also called *parameter*. (3) In the shell, a parameter passed to a utility as the equivalent of a

single string in the *argv* array created by one of the *exec* functions. An argument is one of the options, option-arguments, or operands following the command name. *X/Open*.

argument declaration. See *parameter declaration*.

arithmetic object. (1) An integral object, a bit field, or floating-point object. (2) A real object or objects having the type float, double, or long double.

array. In programming languages, an aggregate that consists of data objects with identical attributes, each of which may be uniquely referenced by subscripting. *IBM*.

array element. A data item in an array. *IBM*.

ASCII. See *American National Standard Code for Information Interchange*.

Assembler H. An IBM licensed program. Translates symbolic assembler language into binary machine language.

assembler language. A source language that includes symbolic language statements in which there is a one-to-one correspondence with the instruction formats and data formats of the computer. *IBM*.

assembler user exit. In the OS/390 Language Environment a routine to tailor the characteristics of an enclave prior to its establishment.

assignment expression. An expression that assigns the value of the right operand expression to the left operand variable and has as its value the value of the right operand. *IBM*.

atexit list. A list of actions specified in the OS/390 C/C++ *atexit()* function that occur at normal program termination.

auto storage class specifier. A specifier that enables the programmer to define a variable with automatic storage; its scope restricted to the current block.

automatic call library. Contains modules that are used as secondary input to the prelinker or the binder to resolve external symbols left undefined after all the primary input has been processed.

The automatic call library can contain:

- Object modules, with or without binder control statements
- Load modules
- OS/390 C/C++ run-time routines (SCEELKED)

automatic library call. The process in which control sections are processed by the binder or loader to

resolve references to members of partitioned data sets. *IBM*.

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*.

B

background process. (1) A process that does not require operator intervention but can be run by the computer while the workstation is used to do other work. *IBM*. (2) A mode of program execution in which the shell does not wait for program completion before prompting the user for another command. *IBM*. (3) A process that is a member of a background process group. *X/Open*. *ISO.1*.

background process group. Any process group, other than a foreground process group, that is a member of a session that has established a connection with a controlling terminal. *X/Open*. *ISO.1*.

backslash. The character \. This character is named <backslash> in the portable character set.

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. The use of existing classes for implementing new classes.

binary expression. An expression containing two operands and one operator.

binary stream. (1) An ordered sequence of untranslated characters. (2) A sequence of characters that corresponds on a one-to-one basis with the characters in the file. No character translation is performed on binary streams. *IBM*.

bind. To combine one or more control sections or program modules into a single program module, resolving references between them, or to assign virtual storage addresses to external symbols.

binder. The DFSMS/MVS program that processes the output of language translators and compilers into an executable program (load module or program object). It replaces the linkage editor and batch loader in the MVS/ESA or OS/390 operating system.

bit field. A member of a structure or union that contains a specified number of bits. *IBM*.

bitwise operator. An operator that manipulates the value of an object at the bit level.

blank character. (1) A graphic representation of the space character. *ANSI/ISO.* (2) A character that represents an empty position in a graphic character string. *ISO Draft.* (3) One of the characters that belong to the *blank* character class as defined via the *LC_CTYPE* category in the current locale. In the *POSIX* locale, a blank character is either a tab or a space character. *X/Open.*

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.

block statement. In the C or C++ languages, a group of data definitions, declarations, and statements appearing between a left brace and a right brace that are processed as a unit. The block statement is considered to be a single C or C++ statement. *IBM.*

boundary alignment. The position in main storage of a fixed-length field, such as a halfword or doubleword, on a byte-level boundary for that unit of information. *IBM.*

braces. The characters { (left brace) and } (right brace), also known as *curly braces*. When used in the phrase “enclosed in (curly) braces” 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-brace> and <right-brace> are used. *X/Open.*

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.*

break statement. A C or C++ control statement that contains the keyword “break” and a semicolon. *IBM.* It is used to end an iterative or a switch statement by exiting from it at any point other than the logical end. Control is passed to the first statement after the iteration or switch statement.

built-in. (1) A function that the compiler will automatically inline instead of making the function call, unless the programmer specifies not to inline. (2) In programming languages, pertaining to a language object that is declared by the definition of the programming language; for example, the built-in

function *SIN* in *PL/I*, the predefined data type *INTEGER* in *FORTRAN.* *ISO-JTC1.* Synonymous with *predefined.* *IBM.*

byte-oriented stream. See *orientation of a stream.*

C

C library. A system library that contains common C language subroutines for file access, string operators, character operations, memory allocation, and other functions. *IBM.*

C or C++ language statement. A C or C++ language statement contains zero or more expressions. A block statement begins with a { (left brace) symbol, ends with a } (right brace) symbol, and contains any number of statements.

All C or C++ language statements, except block statements, end with a ; (semicolon) symbol.

c89 utility. A utility used to compile and bind an OS/390 UNIX application program from the OS/390 shell.

C++ class library. A collection of C++ classes.

C++ library. A system library that contains common C++ language subroutines for file access, memory allocation, and other functions.

callable services. A set of services that can be invoked by a OS/390 Language Environment-conforming high level language using the conventional OS/390 Language Environment-defined call interface, and usable by all programs sharing the OS/390 Language Environment conventions.

Use of these services helps to decrease an application's dependence on the specific form and content of the services delivered by any single operating system.

call chain. A trace of all active routines and subroutines.

caller. A routine that calls another routine.

cancelability point. A specific point within the current thread that is enabled to solicit cancel requests. This is accomplished using the *pthread_testintr()* function.

carriage-return character. A character that in the output stream indicates that printing should start at the beginning of the same physical line in which the carriage-return character occurred. The carriage-return is the character designated by '\r' in the C and C++ languages. It is unspecified whether this character is the exact sequence transmitted to an output device by the

system to accomplish the movement to the beginning of the line. *X/Open*.

case clause. In a C or C++ switch statement, a CASE label followed by any number of statements.

case label. The word case followed by a constant expression and a colon. When the selector evaluates the value of the constant expression, the statements following the case label are processed.

cast expression. A cast expression explicitly converts its operand to a specified arithmetic, scalar, or class type.

cast operator. The cast operator is used for explicit type conversions.

cataloged procedures. A set of control statements placed in a library and retrievable by name. *IBM*.

catch block. A block associated with a try block that receives control when an exception matching its argument is thrown.

char specifier. A char is a built-in data type. In the C++ language, char, signed char, and unsigned char are all distinct data types.

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. *ANSI/ISO*. (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* (multibyte character), where a single-byte character is a special case of the multibyte 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 class. A named set of characters sharing an attribute associated with the name of the class. The classes and the characters that they contain are dependent on the value of the LC_CTYPE category in the current locale. *X/Open*.

character constant. (1) A constant with a character value. *IBM*. (2) A string of any of the characters that can be represented, usually enclosed in apostrophes. *IBM*. (3) In some languages, a character enclosed in apostrophes. *IBM*.

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 special file. (1) A special file that provides access to an input or output device. The character interface is used for devices that do not use block I/O. *IBM*. (2) A file that refers to a device. One specific type of character special file is a terminal device file. *X/Open*. *ISO.1*.

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 is not a child.

child enclave. The *nested enclave* created as a result of certain commands being issued from a *parent enclave*.

CICS (Customer Information Control System). Pertaining to an IBM licensed program that enables transactions entered at remote terminals to be processed concurrently by user-written application programs. It includes facilities for building, using, and maintaining databases. *IBM*.

CICS destination control table. See *DCT*.

CICS translator. A routine that accepts as input an application containing EXEC CICS commands and produces as output an equivalent application in which each CICS command has been translated into the language of the source.

class. (1) A C++ aggregate that may contain functions, types, and user-defined operators in addition to data. Classes may be defined hierarchically, allowing one class to be derived from another, and may restrict access to its members. (2) A user-defined data type. A class data type can contain both data representations (data members) and functions (member functions).

class key. One of the C++ keywords: class, struct and union.

class library. A collection of classes.

class member operator. An operator used to access class members through class objects or pointers to class objects. The class member operators are:

. -> .* ->*

class name. A unique identifier of a class type that becomes a reserved word within its scope.

class scope. An indication that a name of a class can be used only in a member function of that class.

class tag. Synonym for *class name*.

class template. A blueprint describing how a set of related classes can be constructed.

client program. A program that uses a class. The program is said to be a *client* of the class.

CLIST. A programming language that typically executes a list of TSO commands.

CLLE (COBOL Load List Entry). Entry in the load list containing the name of the program and the load address.

COBCOM. Control block containing information about a COBOL partition.

COBOL (common business-oriented language). A high-level language, based on English, that is primarily used for business applications.

COBOL Load List Entry. See *CLLE*.

COBVEC. COBOL vector table containing the address of the library routines.

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. *ANSI/ISO*.

code element set. (1) The result of applying a code to all elements of a coded set, for example, all the three-letter international representations of airport names. *ISO Draft*. (2) The result of applying rules that map a numeric code value to each element of a character set. An element of a character set may be related to more than one numeric code value but the reverse is not true. However, for state-dependent encodings the relationship between numeric code values to elements of a character set may be further controlled by state information. The character set may contain fewer elements than the total number of possible numeric code values; that is, some code values may be unassigned. *X/Open*. (3) Synonym for codeset.

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, assignment of characters and meanings to 128 code points for a 7-bit code. (2) A particular assignment of hexadecimal identifiers to graphic characters.

code point. (1) A 1-byte code representing one of 256 potential characters. (2) An identifier in an alert description that represents a short unit of text. The code point is replaced with the text by an alert display program.

codeset. Synonym for code element set. *IBM*.

collating element. The smallest entity used to determine the logical ordering of character or wide-character strings. A collating element consists of either a single character, or two or more characters collating as a single entity. The value of the LC_COLLATE category in the current locale determines the current set of collating elements. *X/Open*.

collating sequence. (1) A specified arrangement used in sequencing. *ISO-JTC1*. *ANSI/ISO*. (2) An ordering assigned to a set of items, such that any two sets in that assigned order can be collated. *ANSI/ISO*. (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.

collation. The logical ordering of character or wide-character strings according to defined precedence rules. These rules identify a collation sequence between the collating elements, and such additional rules that can be used to order strings consisting of multiple collating elements. *X/Open*.

collection. (1) An abstract class without any ordering, element properties, or key properties. All abstract classes are derived from collection. (2) In a general sense, an implementation of an abstract data type for storing elements.

Collection Class Library. A set of classes that provide basic functions for collections, and can be used as base classes.

column position. A unit of horizontal measure related to characters in a line.

It is assumed that each character in a character set has an intrinsic column width independent of any output device. Each printable character in the portable character set has a column width of one. The standard utilities, when used as described in this document set, assume that all characters have integral column widths. The column width of a character is not necessarily

related to the internal representation of the character (numbers of bits or bytes).

The column position of a character in a line is defined as one plus the sum of the column widths of the preceding characters in the line. Column positions are numbered starting from 1. *X/Open*.

comma expression. An expression that contains two operands separated by a comma. Although the compiler evaluates both operands, the value of the expression is the value of the right operand. If the left operand produces a value, the compiler discards this value. Typically, the left operand of a comma expression is used to produce side effects.

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.

command processor parameter list (CPPL). The format of a TSO parameter list. When a TSO terminal monitor application attaches a command processor, register 1 contains a pointer to the CPPL, containing addresses required by the command processor.

COMMAREA. A communication area made available to applications running under CICS.

Common Business-Oriented Language. See *COBOL*.

common expression elimination. Duplicated expressions are eliminated by using the result of the previous expression. This includes intermediate expressions within expressions.

compilation unit. (1) A portion of a computer program sufficiently complete to be compiled correctly. *IBM*. (2) A single compiled file and all its associated include files. (3) An independently compilable sequence of high-level language statements. Each high-level language product has different rules for what makes up a compilation unit.

complete class name. The complete qualification of a nested class name including all enclosing class names.

Complex Mathematics library. A C++ class library that provides the facilities to manipulate complex numbers and perform standard mathematical operations on them.

computational independence. No data modified by either a main task program or a parallel function is examined or modified by a parallel function that might be running simultaneously.

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 OS/390 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).

condition handler. A user-written condition handler or language-specific condition handler (such as a PL/I ON-unit or OS/390 C/C++ `signal()` function call) invoked by the OS/390 C/C++ *condition manager* to respond to conditions.

condition manager. Manages conditions in the common execution environment by invoking various user-written and language-specific *condition handlers*.

condition token. In the OS/390 Language Environment, a data type consisting of 12 bytes (96 bits). The condition token contains structured fields that indicate various aspects of a condition including the severity, the associated message number, and information that is specific to a given instance of the condition.

const. (1) An attribute of a data object that declares the object cannot be changed. (2) A keyword that allows you to define a variable whose value does not change.

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*.

constant propagation. An optimization technique where constants used in an expression are combined and new ones are generated. Mode conversions are done to allow some intrinsic functions to be evaluated at compile time.

constructed reentrancy. The attribute of applications that contain external data and require additional processing to make them reentrant. Contrast with *natural reentrancy*.

constructor. A special C++ class member function that has the same name as the class and is used to create an object of that class.

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.*

control statement. (1) In programming languages, a statement that is used to alter the continuous sequential execution of statements; a control statement may be a conditional statement, such as IF, or an imperative statement, such as STOP. *ISO Draft.* (2) A statement that changes the path of execution.

controlling process. The session leader that establishes the connection to the controlling terminal. If the terminal ceases to be a controlling terminal for this session, the session leader ceases to be the controlling process. *X/Open. ISO.1.*

controlling terminal. A terminal that is associated with a session. Each session may have at most one controlling terminal associated with it, and a controlling terminal is associated with exactly one session. Certain input sequences from the controlling terminal cause signals to be sent to all processes in the process group associated with the controlling terminal. *X/Open. ISO.1.*

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 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 descriptor. A per-process unique value used to identify an open codeset conversion. *X/Open.*

conversion function. A member function that specifies a conversion from its class type to another type.

coordinated universal time (UTC). Synonym for Greenwich Mean Time (GMT). See *GMT.*

copy constructor. A constructor that copies a class object of the same class type.

Cross System Product. See *CSP.*

CSP (Cross System Product). A set of licensed programs designed to permit the user to develop and run applications using independently defined maps (display and printer formats), data items (records, working storage, files, and single items), and processes (logic). The Cross System Product set consists of two parts: Cross System Product/Application Development (CSP/AD) and Cross System Product/Application Execution (CSP/AE). *IBM.*

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 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.* (3) In the OS/390 UNIX environment, 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.

Customer Information Control System. See *CICS.*

D

data abstraction. A data type with a private representation and a public set of operations (functions or operators) which restrict access to that data type to that set of operations. The C++ language uses the concept of classes to implement data abstraction.

DATABASE 2. Pertaining to an IBM relational database.

data definition (DD). (1) In the C and C++ languages, a definition that describes a data object, reserves storage for a data object, and can provide an initial value for a data object. A data definition appears outside a function or at the beginning of a block statement. *IBM.* (2) A program statement that describes the features of, specifies relationships of, or establishes context of, data. *ANSI/ISO.* (3) A statement that is stored in the environment and that externally identifies a file and the attributes with which it should be opened.

data definition name. See *ddname.*

data definition statement. See *DD statement.*

data member. The smallest possible piece of complete data. Elements are composed of data members.

data object. (1) A storage area used to hold a value. (2) Anything that exists in storage and on which

operations can be performed, such as files, programs, classes, or arrays. (3) In a program, an element of data structure, such as a file, array, or operand, that is needed for the execution of a program and that is named or otherwise specified by the allowable character set of the language in which a program is coded. *IBM.*

data set. Under MVS, a named collection of related data records that is stored and retrieved by an assigned name.

data stream. A continuous stream of data elements being transmitted, or intended for transmission, in character or binary-digit form, using a defined format. *IBM.*

data structure. The internal data representation of an implementation.

data type. The properties and internal representation that characterize data.

Data Window Services (DWS). Services provided as part of the Callable Services Library that allow manipulation of data objects such as VSAM linear data sets and temporary data objects known as *TEMPSPACE*.

DBCS (double-byte character set). 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, the typing, display, and printing of DBCS characters requires hardware and programs that support DBCS. *IBM.*

DCT (destination control table). A table that contains an entry for each extrapartition, inrapartition, and indirect destination. Extrapartition entries address data sets external to the CICS region. Inrapartition destination entries contain the information required to locate the queue in the inrapartition data set. Indirect destination entries contain the information required to locate the queue in the inrapartition data set.

ddname (data definition name). (1) The logical name of a file within an application. The ddname provides the means for the logical file to be connected to the physical file. (2) The part of the data definition before the equal sign. It is the name used in a call to *fopen* or *freopen* to refer to the data definition stored in the environment.

DD statement (data definition statement). (1) In MVS, serves as the connection between the logical name of a file and the physical name of the file. (2) A

job control statement that defines a file to the operating system, and is a request to the operating system for the allocation of input/output resources.

dead code elimination. A process that eliminates code that exists for calculations that are not necessary. Code may be designated as dead by other optimization techniques.

dead store elimination. A process that eliminates unnecessary storage use in code. A store is deemed unnecessary if the value stored is never referenced again in the code.

decimal constant. (1) A numerical data type used in standard arithmetic operations. (2) A number containing any of the digits 0 through 9. *IBM.*

decimal overflow. A condition that occurs when one or more nonzero digits are lost because the destination field in a decimal operation is too short to contain the results.

declaration. (1) In the C and C++ languages, a description that makes an external object or function available to a function or a block statement. *IBM.* (2) Establishes the names and characteristics of data objects and functions used in a program.

declarator. Designates a data object or function declared. Initializations can be performed in a declarator.

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 clause. In the C or C++ languages, within a switch statement, the keyword *default* followed by a colon, and one or more statements. When the conditions of the specified case labels in the switch statement do not hold, the default clause is chosen. *IBM.*

default constructor. A constructor that takes no arguments, or, if it takes arguments, all its arguments have default values.

default initialization. The initial value assigned to a data object by the compiler if no initial value is specified by the programmer.

default locale. (1) The C locale, which is always used when no selection of locale is performed. (2) A system default locale, named by locale-related environmental variables.

define directive. A preprocessor statement that directs the preprocessor to replace an identifier or macro invocation with special code.

define statement. A preprocessor statement that causes the preprocessor to replace an identifier or macro call with specified code. *IBM.*

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 *new*.

demangling. The conversion of mangled names back to their original source code names. During C++ compilation, identifiers such as function and static class member names are mangled (encoded) with type and scoping information to ensure type-safe linkage. These mangled names appear in the object file and the final executable file. Demangling (decoding) converts these names back to their original names to make program debugging easier. See also *mangling*.

denormal. Pertaining to a number with a value so close to 0 that its exponent cannot be represented normally. The exponent can be represented in a special way at the possible cost of a loss of significance.

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.

dereference. In the C and C++ languages, the application of the unary operator * to a pointer to access the object the pointer points to. Also known as *indirection*.

derivation. In the C++ language, to derive a class, called a derived class, from an existing class, called a base class.

derived class. A class that inherits from a base class. All members of the base class become members of the derived class. You can add additional data members and member functions to the derived class. A derived class object can be manipulated as if it is a base class object. The derived class can override virtual functions of the base class.

descriptor. PL/I control block that holds information such as string lengths, array subscript bounds, and area sizes, and is passed from one PL/I routine to another during run time.

destination control table. See *DCT*.

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, freeing storage that was allocated when the object was created. A destructor has no arguments and no return type.

detach state attribute. An attribute associated with a thread attribute object. This attribute has two possible values:

- 0** Undetached. An undetached thread keeps its resources after termination of the thread.
- 1** Detached. A detached thread has its resources freed by the system after termination.

device. A computer peripheral or an object that appears to the application as such. *X/Open. ISO.1.*

difference. For two sets A and B, the difference (A-B) is the set of all elements in A but not in B. For bags, there is an additional rule for duplicates: If bag P contains an element *m* times and bag Q contains the same element *n* times, then, if $m > n$, the difference contains that element $m-n$ times. If $m \leq n$, the difference contains that element zero times.

digraph. A combination of two keystrokes used to represent unavailable characters in a C++ source program. Digraphs are read as tokens during the preprocessor phase.

directory. A type of file containing the names and controlling information for other files or other directories. *IBM.*

Direct-to-SOM (DTS). (1) Term applied to the method by which the OS/390 C++ compiler converts existing C++ classes to SOM classes. (2) Term applied to a class that has been converted to SOM by the OS/390 C++ compiler.

disabled signal. Synonym for *enabled signal*.

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.*

do statement. In the C and C++ compilers, a looping statement that contains the keyword "do," followed by a statement (the action), the keyword "while," and an expression in parentheses (the condition). *IBM.*

dot. The file name consisting of a single dot character (.). *X/Open. ISO.1.*

double-byte character set. See *DBCS*.

double-precision. Pertaining to the use of two computer words to represent a number in accordance with the required precision. *ISO-JTC1. ANSI/ISO.*

double-quote. The character `"`, also known as *quotation mark*. *X/Open.*

This character is named `<quotation-mark>` in the portable character set.

doubleword. A contiguous sequence of bits or characters that comprises two computer words and is capable of being addressed as a unit. *IBM.*

dynamic. Pertaining to an operation that occurs at the time it is needed rather than at a predetermined or fixed time. *IBM.*

dynamic allocation. Assignment of system resources to a program when the program is executed rather than when it is loaded into main storage. *IBM.*

dynamic binding. The act of resolving references to external variables and functions at run time.

dynamic link library (DLL). A file containing executable code and data bound to a program at run time. The code and data in a dynamic link library can be shared by several applications simultaneously. Compiling code with the DLL option does not mean that the produced executable will be a DLL. To create a DLL, use `#pragma export` or the `EXPORTALL` compiler option.

DSA (dynamic storage area). An area of storage obtained during the running of an application that consists of a register save area and an area for automatic data, such as program variables. DSAs are generally allocated within Language Environment-managed stack segments. DSAs are added to the stack when a routine is entered and removed upon exit in a last in, first out (LIFO) manner. In Language Environment, a DSA is known as a stack frame.

dynamic storage. Synonym for *automatic storage*.

dynamic storage area. See DSA

E

EBCDIC. See *extended binary-coded decimal interchange code*.

effective group ID. An attribute of a process that is used in determining various permissions, including file access permissions. This value is subject to change during the process lifetime, as described in the *exec* family of functions and `setgid()`. *X/Open. ISO.1.*

effective user ID. (1) The user ID associated with the last authenticated user or the last `setuid()` program. It is equal to either the real or the saved user ID. (2) The current user ID, but not necessarily the user's login ID; for example, a user logged in under a login ID may change to another user's ID. The ID to which the user changes becomes the effective user ID until the user switches back to the original login ID. All discretionary access decisions are based on the effective user ID. *IBM.* (3) An attribute of a process that is used in determining various permissions, including file access permissions. This value is subject to change during the process lifetime, as described in *exec* and `setuid()`. *X/Open. ISO.1.*

elaborated type specifier. A specifier typically used in an incomplete class declaration to qualify types that are otherwise hidden.

element. The component of an array, subrange, enumeration, or set.

element equality. A relation that determines if two elements are equal.

element occurrence. A single instance of an element in a collection. In a unique collection, element occurrence is synonymous with element value.

element value. All the instances of an element with a particular value in a collection. In a nonunique collection, an element value may have more than one occurrence. In a unique collection, element value is synonymous with element occurrence.

else clause. The part of an if statement that contains the word *else*, followed by a statement. The *else* clause provides an action that is started when the if condition evaluates to a value of zero (false). *IBM.*

empty line. A line consisting of only a new-line character. *X/Open.*

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.1.*

enabled signal. The occurrence of an enabled signal results in the default system response or the execution of an established signal handler. If disabled, the occurrence of the signal is ignored.

encapsulation. Hiding the internal representation of data objects and implementation details of functions from the client program. This enables the end user to focus on the use of data objects and functions without having to know about their representation or implementation.

enclave. In the Language Environment for MVS and VM, an independent collection of routines, one of which is designated as the main routine. An enclave is roughly analogous to a program or run unit.

enqueue. An operation that adds an element as the last element to a queue.

entry point. In assembler language, the address or label of the first instruction that is executed when a routine is entered for execution.

enumeration constant. In the C or C++ language, an identifier, with an associated integer value, defined in an enumerator. An enumeration constant may be used anywhere an integer constant is allowed. *IBM.*

enumeration data type. (1) In the Fortran, C, and C++ language, a data type that represents a set of values that a user defines. *IBM.* (2) A type that represents integers and a set of enumeration constants. Each enumeration constant has an associated integer value.

enumeration tag. In the C and C++ language, the identifier that names an enumeration data type. *IBM.*

enumeration type. An enumeration type defines a set of enumeration constants. In the C++ language, an enumeration type is a distinct data type that is not an integral type.

enumerator. In the C and C++ language, an enumeration constant and its associated value. *IBM.*

equivalence class. (1) A grouping of characters that are considered equal for the purpose of collation; for example, many languages place an uppercase character in the same equivalence class as its lowercase form, but some languages distinguish between accented and unaccented character forms for the purpose of collation. *IBM.* (2) A set of collating elements with the same primary collation weight.

Elements in an equivalence class are typically elements that naturally group together, such as all accented letters based on the same base letter.

The collation order of elements within an equivalence class is determined by the weights assigned on any subsequent levels after the primary weight. *X/Open.*

escape sequence. (1) A representation of a character. An escape sequence contains the \ symbol followed by one of the characters: a, b, f, n, r, t, v, ' , ", x, \, or followed by one or more octal or hexadecimal digits. (2) A sequence of characters that represent, for example, nonprinting characters, or the exact code point value to be used to represent variant and nonvariant characters regardless of code page. (3) In the C and C++ language, an escape character followed by one or

more characters. The escape character indicates that a different code, or a different coded character set, is used to interpret the characters that follow. Any member of the character set used at runtime can be represented using an escape sequence. (4) A character that is preceded by a backslash character and is interpreted to have a special meaning to the operating system. (5) A sequence sent to a terminal to perform actions such as moving the cursor, changing from normal to reverse video, and clearing the screen. Synonymous with multibyte control. *IBM.*

exception. (1) 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). (2) In programming languages, an abnormal situation that may arise during execution, that may cause a deviation from the normal execution sequence, and for which facilities exist in a programming language to define, raise, recognize, ignore, and handle it; for example, (ON-) condition in PL/I, exception in ADA. *ISO-JTC1.*

executable. A load module or program object which has yet to be loaded into memory for execution.

executable file. A regular file acceptable as a new process image file by the equivalent of the *exec* family of functions, and thus usable as one form of a utility. The standard utilities described as compilers can produce executable files, but other unspecified methods of producing executable files may also be provided. The internal format of an executable file is unspecified, but a conforming application cannot assume an executable file is a text file. *X/Open.*

exception handler. (1) Exception handlers are catch blocks in C++ applications. Catch blocks catch exceptions when they are thrown from a function enclosed in a try block. Try blocks, catch blocks, and throw expressions are the constructs used to implement formal exception handling in C++ applications. (2) A set of routines used to detect deadlock conditions or to process abnormal condition processing. An exception handler allows the normal running of processes to be interrupted and resumed. *IBM.*

executable file. A regular file acceptable as a new process image file by the equivalent of the *exec* family of functions, and thus usable as one form of a utility. The standard utilities described as compilers can produce executable files, but other unspecified methods of producing executable files may also be provided. The internal format of an executable file is unspecified, but a conforming application cannot assume an executable file is a text file. *X/Open.*

executable program. A program that has been link-edited and therefore can be run in a processor. *IBM.*

extended binary-coded data interchange code (EBCDIC). A coded character set of 256 8-bit characters. *IBM.*

extension. (1) An element or function not included in the standard language. (2) File name extension.

external data definition. A description of a variable appearing outside a function. It causes the system to allocate storage for that variable and makes that variable accessible to all functions that follow the definition and are located in the same file as the definition. *IBM.*

extern storage class specifier. A specifier that enables the programmer to declare objects and functions that several source files can use.

F

feature test macro (FTM). A macro (`#define`) used to determine whether a particular set of features will be included from a header. *X/Open. ISO.1.*

FIFO special file. A type of file with the property that data written to such a file is read on a first-in-first-out basis. Other characteristics of FIFOs are described in `open()`, `read()`, `write()`, and `lseek()`. *X/Open. ISO.1.*

file access permissions. The standard file access control mechanism uses the file permission bits. The bits are set at the time of file creation by functions such as `open()`, `creat()`, `mkdir()`, and `mkfifo()` and can be changed by `chmod()`. The bits are read by `stat()` or `fstat()`. *X/Open.*

file descriptor. (1) A small positive integer that the system uses instead of the file name to identify an open file. *IBM.* (2) A per-process unique, non-negative integer used to identify an open file for the purpose of file access. *ISO.1.*

The value of a file descriptor is from zero to `{OPEN_MAX}`—which is defined in `<limits.h>`. A process can have no more than `{OPEN_MAX}` file descriptors open simultaneously. File descriptors may also be used to implement directory streams. *X/Open.*

file mode. An object containing the *file mode bits* and file type of a file, as described in `<sys/stat.h>`. *X/Open.*

file mode bits. A file's file permission bits, set-user-ID-on-execution bit (`S_ISUID`) and set-group-ID-on-execution bit (`S_ISGID`). *X/Open.*

file permission bits. Information about a file that is used, along with other information, to determine if a process has read, write, or execute/search permission to a file. The bits are divided into three parts: owner, group, and other. Each part is used with the

corresponding file class of process. These bits are contained in the file mode, as described in `<sys/stat.h>`. The detailed usage of the file permission bits is described in *file access permissions. X/Open. ISO.1.*

file scope. A name declared outside all blocks and classes has file scope and can be used after the point of declaration in a source file.

filter. A command whose operation consists of reading data from standard input or a list of input files and writing data to standard output. Typically, its function is to perform some transformation on the data stream. *X/Open.*

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.

float constant. (1) A constant representing a nonintegral number. (2) A number containing a decimal point, an exponent, or both a decimal point and an exponent. The exponent contains an `e` or `E`, an optional sign (+ or -), and one or more digits (0 through 9). *IBM.*

for statement. A looping statement that contains the word *for* followed by a list of expressions enclosed in parentheses (the condition) and a statement (the action). Each expression in the parenthesized list is separated by a semicolon. You can omit any of the expressions, but you cannot omit the semicolons.

foreground process. (1) A process that must run to completion before another command is issued. The foreground process is in the foreground process group, which is the group that receives the signals generated by a terminal. *IBM.* (2) A process that is a member of a foreground process group. *X/Open. ISO.1.*

foreground process group. (1) The group that receives the signals generated by a terminal. *IBM.* (2) A process group whose member processes have certain privileges, denied to processes in background process groups, when accessing their controlling terminal. Each session that has established a connection with a controlling terminal has exactly one process group of the session as the foreground process group of that controlling terminal. *X/Open. ISO.1.*

foreground process group ID. The process group ID of the foreground process group. *X/Open. ISO.1.*

form-feed character. A character in the output stream that indicates that printing should start on the next page of an output device. The formfeed is the character designated by `'f'` in the C and C++ language. If the formfeed is not the first character of an output line, the

result is unspecified. It is unspecified whether this character is the exact sequence transmitted to an output device by the system to accomplish the movement to the next page. *X/Open*.

forward declaration. A declaration of a class or function made earlier in a compilation unit, so that the declared class or function can be used before it has been defined.

freestanding application. (1) An application that is created to run without the run-time environment or library with which it was developed. (2) An OS/390 C/C++ application that does not use the services of the dynamic OS/390 C/C++ run-time library or of the Language Environment. Under OS/390 C support, this ability is a feature of the System Programming C support.

free store. Dynamically allocated memory. New and delete are used to allocate and deallocate free store.

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 another class and uses the keyword friend as a prefix to the class. For example, the following source code makes all the functions and data in class you friends of class me:

```
class me {  
    friend class you;  
    // ...  
};
```

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 other class with the prefix friend.

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 declarator. The part of a function definition that names the function, provides additional information about the return value of the function, and lists the function parameters. *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.

function scope. Labels that are declared in a function have function scope and can be used anywhere in that function.

function template. Provides a blueprint describing how a set of related individual functions can be constructed.

G

Generalization. Refers to a class, function, or static data member which derives its definition from a template. An instantiation of a template function would be a generalization.

generic class. Synonym for *class templates*.

global. Pertaining to information available to more than one program or subroutine. *IBM*.

global scope. Synonym for *file scope*.

global variable. A symbol defined in one program module that is used in other independently compiled program modules.

GMT (Greenwich Mean Time). The solar time at the meridian of Greenwich, formerly used as the prime basis of standard time throughout the world. GMT has been superseded by coordinated universal time (UTC).

graphic character. (1) A visual representation of a character, other than a control character, that is normally produced by writing, printing, or displaying. *ISO Draft*. (2) A character that can be displayed or printed. *IBM*.

Graphical Data Display Manager (GDDM). Pertaining to an IBM licensed program that provides a group of routines that allows pictures to be defined and displayed procedurally through function routines that correspond to graphic primitives. *IBM*.

Greenwich Mean Time. See GMT.

group ID. (1) A non-negative integer that is used to identify a group of system users. Each system user is a member of at least one group. When the identity of a group is associated with a process, a group ID value is referred to as a real group ID, an effective group ID, one of the supplementary group IDs or a saved set-group-ID. *X/Open*. (2) A non-negative integer,

which can be contained in an object of type *gid_t*, that is used to identify a group of system users. *ISO.1*.

H

halfword. A contiguous sequence of bits or characters that constitutes half a computer word and can be addressed as a unit. *IBM*.

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. (1) 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. (2) A table of information that is accessed by way of a shortened search key (that hash value). Using a hash table minimizes average search time.

header file. A text file that contains declarations used by a group of functions, programs, or users.

heap storage. An area of storage used for allocation of storage whose lifetime is not related to the execution of the current routine. The heap consists of the initial heap segment and zero or more increments.

hexadecimal constant. A constant, usually starting with special characters, that contains only hexadecimal digits. Three examples for the hexadecimal constant with value 0 would be '*\x00*', '*0x0*', or '*0X00*'.

hyperspace memory file. An IBM file used under MVS to deal with memory files as large as 2 gigabytes. *IBM*.

hooks. Instructions inserted into a program by a compiler at compile-time. Using hooks, you can set break-points to instruct the Debug Tool to gain control of the program at selected points during its execution.

hybrid code. Program statements that have not been internationalized with respect to code page, especially where data constants contain variant characters. Such statements can be found in applications written in older implementations of MVS, which required syntax statements to be written using code page IBM-1047 exclusively. Such applications cannot be converted from one code page to another using *iconv()*.

I

I18N. Abbreviation for *internationalization*.

identifier. (1) One or more characters used to identify or name a data element and possibly to indicate certain properties of that data element. *ANSI/ISO*. (2) In programming languages, a token that names a data

object such as a variable, an array, a record, a subprogram, or a function. *ANSI/ISO*. (3) A sequence of letters, digits, and underscores used to identify a data object or function. *IBM*.

if statement. A conditional statement that contains the keyword *if*, followed by an expression in parentheses (the condition), a statement (the action), and an optional *else* clause (the alternative action). *IBM*.

ILC (interlanguage call). A function call made by one language to a function coded in another language. Interlanguage calls are used to communicate between programs written in different languages.

ILC (interlanguage communication). The ability of routines written in different programming languages to communicate. ILC support enables the application writer to readily build applications from component routines written in a variety of languages.

implementation-defined behavior. Application behavior that is not defined by the standards. The implementing compiler and library defines this behavior when a program contains correct program constructs or uses correct data. Programs that rely on implementation-defined behavior may behave differently on different C or C++ implementations. Refer to the OS/390 C/C++ books that are listed in "IBM OS/390 C/C++ and Related Publications" on page *lii* for information about implementation-defined behavior in the OS/390 C/C++ environment. Contrast with *unspecified behavior* and *undefined behavior*.

IMS (Information Management System). Pertaining to an IBM database/data communication (DB/DC) system that can manage complex databases and networks. *IBM*.

include directive. A preprocessor directive that causes the preprocessor to replace the statement with the contents of a specified file.

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. Until a class is fully declared, or defined, you can only use the class name where the size of the class is not required. Typically an incomplete class declaration is used as a forward declaration.

incomplete type. A type that has no value or meaning when it is first declared. There are three incomplete types: void, arrays of unknown size and structures and unions of unspecified content. A void type can never be

completed. Arrays of unknown size and structures or unions of unspecified content can be completed in further declarations.

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 to which the pointer points.

indirection class. Synonym for *reference class*.

inheritance. A technique that allows the use of an existing class as the base for creating other classes.

initial heap. The OS/390 C/C++ heap controlled by the HEAP runtime option and designated by a `heap_id` of 0. The initial heap contains dynamically allocated user data.

initializer. An expression used to initialize data objects. The C++ language, supports the following types of initializers:

- An expression followed by an assignment operator that is used to initialize fundamental data type objects or class objects that contain copy constructors.
- A parenthesized expression list that is used to initialize base classes and members that use constructors.

Both the C and C++ languages support an expression enclosed in braces (`{ }`), that used to initialize aggregates.

inlined function. A function whose actual code replaces a function call. A function that is both declared and defined in a class definition is an example of an inline function. Another example is one which you explicitly declared inline by using the keyword `inline`. Both member and nonmember functions can be inlined.

input stream. A sequence of control statements and data submitted to a system from an input unit. Synonymous with input job stream, job input stream. *IBM.*

instance. An object-oriented programming term synonymous with object. An instance is a particular instantiation of a data type. It is simply a region of storage that contains a value or group of values. For example, if a class `box` is previously defined, two instances of a class `box` could be instantiated with the declaration: `box box1, box2;`

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.

instruction scheduling. An optimization technique that reorders instructions in code to minimize execution time.

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 System Productivity Facility. See *ISPF*.

interlanguage call. See *ILC (interlanguage call)*.

interlanguage communication. See *ILC (interlanguage communication)*.

internationalization. The capability of a computer program to adapt to the requirements of different native languages, local customs, and coded character sets. *X/Open.*

Synonymous with *I18N*.

interoperability. The capability to communicate, execute programs, or transfer data among various functional units in a way that requires the user to have little or no knowledge of the unique characteristics of those units.

Interprocedural Analysis. See *IPA*.

interprocess communication. (1) The exchange of information between processes or threads through semaphores, queues, and shared memory. (2) The process by which programs communicate data to each other to synchronize their activities. Semaphores, signals, and internal message queues are common methods of inter-process communication.

I/O Stream library. A class library that provides the facilities to deal with many varieties of input and output.

IPA (Interprocedural Analysis). A process for performing optimizations across compilation units.

ISPF (Interactive System Productivity Facility). An IBM licensed program that serves as a full-screen editor and dialogue manager. Used for writing application programs, it provides a means of generating standard screen panels and interactive dialogues between the application programmer and terminal user. (ISPF)

iteration. The process of repeatedly applying a function to a series of elements in a collection until some condition is satisfied.

J

JCL (job control language). A control language used to identify a job to an operating system and to describe the job's requirement. *IBM.*

job control. A facility that allows users to selectively stop (suspend) the execution of a process and continue (resume) their execution at a later point.

The user typically employs this facility via the interactive interface jointly supplied by the terminal I/O driver and a command interpreter. *X/Open. ISO.1.*

K

keyword. (1) A predefined word reserved for the C and C++ languages, that may not be used as an identifier. (2) A symbol that identifies a parameter in JCL.

kind attribute. An attribute for a mutex attribute object. This attribute's value determines whether the mutex can be locked once or more than once for a thread and whether state changes to the mutex will be reported to the debug interface.

L

label. An identifier within or attached to a set of data elements. *ISO Draft.*

Language Environment. Abbreviated form of IBM Language Environment for MVS and VM. Pertaining to an IBM software product that provides a common runtime environment and runtime services to applications compiled by Language Environment-conforming compilers.

last element. The element visited 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.

late binding. Allowing the system to determine the specific class of the object and invoke the appropriate function implementations at run time. Late binding or dynamic binding hides the differences between a group of related classes from the application program.

leaves. Nodes without children. Synonymous with terminals.

lexically. Relating to the left-to-right order of units.

library. (1) A collection of functions, calls, subroutines, or other data. *IBM.* (2) A set of object modules that can be specified in a link command.

linkage editor. Synonym for linker. The linkage editor has been replaced by the *binder* for the MVS/ESA or OS/390 operating systems. See *binder*.

Linkage. Refers to the binding between a reference and a definition. A function has internal linkage if the function is defined inline as part of the class, is declared with the inline keyword, or is a nonmember function declared with the static keyword. All other functions have external linkage.

linker. A computer program for creating load modules from one or more object modules by resolving cross references among the modules and, if necessary, adjusting addresses. *IBM.*

link pack area (LPA). In MVS, an area of storage containing re-enterable routines from system libraries. Their presence in main storage saves loading time.

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. *ANSI/ISO.*

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. *ANSI/ISO.*

local customs. The conventions of a geographical area or territory for such things as date, time, and currency formats. *X/Open.*

locale. The definition of the subset of a user's environment that depends on language and cultural conventions. *X/Open.*

localization. The process of establishing information within a computer system specific to the operation of

particular native languages, local customs, and coded character sets. *X/Open*.

local scope. A name declared in a block has scope within the block, and can therefore only be used in that block.

Long name. An external name C++ name in an object module, or and external name in an object module created by the C compiler when the `LONGNAME` option is used. Long names are up to 1024 characters long and may contain both upper-case and lower-case characters.

lvalue. An expression that represents a data 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.

macro call. Synonym for *macro*.

macro instruction. Synonym for *macro*.

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`.

makefile. A text file containing a list of your application's parts. The make utility uses makefiles to maintain application parts and dependencies.

make utility. Maintains all of the parts and dependencies for your application. The make utility uses a makefile to keep the parts of your program synchronized. If one part of your application changes, the make utility updates all other files that depend on the changed part. This utility is available under the OS/390 shell and by default, uses the `c89` utility to recompile and bind your application.

mangling. The encoding during compilation of identifiers such as function and variable names to include type and scope information. These mangled names ensure type-safe linkage. See also *demangling*.

manipulator. A value that can be inserted into streams or extracted from streams to affect or query the behavior of the stream.

member. A data object or function in a structure, union, or class. Members can also be classes, enumerations, bit fields, and type names.

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.

method. In the C++ language, a synonym for *member function*.

migrate. To move to a changed operating environment, usually to a new release or version of a system. *IBM*.

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.

multibyte character. A mixture of single-byte characters from a single-byte character set and double-byte characters from a double-byte character set.

multicharacter collating element. A sequence of two or more characters that collate as an entity. For example, in some coded character sets, an accented character is represented by a non-spacing accent, followed by the letter. Other examples are the Spanish elements *ch* and *ll*. *X/Open*.

multiple inheritance. An object-oriented programming technique implemented in the C++ language through derivation, in which the derived class inherits members from more than one base class.

multitasking. A mode of operation that allows concurrent performance, or interleaved execution of two or more tasks. *ISO-JTC1. ANSI/ISO*.

mutex. A flag used by a semaphore to protect shared resources. The mutex is locked and unlocked by threads in a program. A mutex can only be locked by one thread at a time and can only be unlocked by the same thread that locked it. The current owner of a mutex is the thread that it is currently locked by. An unlocked mutex has no current owner.

mutex attribute object. Allows the user to manage the characteristics of mutexes in their application by defining a set of values to be used for the mutex during its creation. A mutex attribute object allows the user to create many mutexes with the same set of characteristics without redefining the same set of characteristics for each mutex created.

mutex object. Used to identify a mutex.

N

name space. A category used to group similar types of identifiers.

named pipe. A FIFO file. Named pipes allow transfer of data between processes in a FIFO manner and synchronization of process execution. Allows processes to communicate even though they do not know what processes are on the other end of the pipe.

natural reentrancy. A program that contains no writable static and requires no additional processing to make it reentrant is considered naturally reentrant.

nested class. A class defined within the scope of another class.

nested enclave. A new enclave created by an existing enclave. The nested enclave that is created must be a new main routine within the process. See also *child enclave* and *parent enclave*.

newline character. A character that in the output stream indicates that printing should start at the beginning of the next line. The newline character is designated by '\n' in the C and C++ language. It is unspecified whether this character is the exact sequence transmitted to an output device by the system to accomplish the movement to the next line. *X/Open*.

nickname. Synonym for alias.

nonprinting character. See *control character*.

null character (NUL). The ASCII or EBCDIC character '\0' 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 statement. A C or C++ statement that consists solely of a semicolon.

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*.

null value. A parameter position for which no value is specified. *IBM*.

null wide-character code. A wide-character code with all bits set to zero. *X/Open*.

number sign. The character #, also known as *pound sign* and *hash sign*. This character is named <number-sign> in the portable character set.

O

object. (1) A region of storage. An object is created when a variable is defined. An object is destroyed when it goes out of scope. (See also *instance*.) (2) In object-oriented design or programming, an abstraction consisting of data and the operations associated with that data. See also *class*. *IBM*. (3) An instance of a class.

object code. Machine-executable instructions, usually generated by a compiler from source code written in a higher level language (such as the C++ language). For programs that must be linked, object code consists of relocatable machine code.

object module. (1) All or part of an object program sufficiently complete for linking. Assemblers and compilers usually produce object modules. *ISO Draft*. (2) A set of instructions in machine language produced by a compiler from a source program. *IBM*.

object-oriented programming. A programming approach based on the concepts of data abstraction and inheritance. Unlike procedural programming techniques, object-oriented programming concentrates not on how something is accomplished, but on what data objects comprise the problem and how they are manipulated.

octal constant. The digit 0 (zero) followed by any digits 0 through 7.

open file. A file that is currently associated with a file descriptor. *X/Open*. *ISO.1*.

operand. An entity on which an operation is performed. *ISO-JTC1*. *ANSI/ISO*.

operating system (OS). Software that controls functions such as resource allocation, scheduling, input/output control, and data management.

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.

operator precedence. In programming languages, an order relation defining the sequence of the application of operators within an expression. *ISO-JTC1*.

orientation of a stream. After application of an input or output function to a stream, it becomes either byte-oriented or wide-oriented. A byte-oriented stream is a stream that had a byte input or output function applied to it when it had no orientation. A wide-oriented stream is a stream that had a wide character input or output function applied to it when it had no orientation. A stream has no orientation when it has been associated with an external file but has not had any operations performed on it.

OS/390 UNIX System Services (OS/390 UNIX). An element of the OS/390 operating system, (formerly known as OpenEdition). OS/390 UNIX includes a POSIX system Application Programming Interface for the C language, a shell and utilities component, and a dbx debugger. All the components conform to IEEE POSIX standards (ISO 9945-1: 1990/IEEE POSIX 1003.1-1990, IEEE POSIX 1003.1a, IEEE POSIX 1003.2, and IEEE POSIX 1003.4a).

overflow. (1) A condition that occurs when a portion of the result of an operation exceeds the capacity of the intended unit of storage. (2) That portion of an operation that exceeds the capacity of the intended unit of storage. *IBM.*

overlay. The technique of repeatedly using the same areas of internal storage during different stages of a program. *ANSI/ISO.*

overloading. An object-oriented programming technique that allows you to redefine functions and most standard C++ operators when the functions and operators are used with class types.

P

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 enclave. The enclave that issues a call to system services or language constructs to create a nested or child enclave. See also *child enclave* and *nested enclave*.

parent process. (1) The program that originates the creation of other processes by means of *spawn* or *exec* function calls. See also *child process*. (2) A process that creates other processes.

parent process ID. (1) An attribute of a new process identifying the parent of the process. The parent process ID of a process is the process ID of its creator, for the lifetime of the creator. After the creator's lifetime has ended, the parent process ID is the process ID of an implementation-dependent system process. *X/Open.* (2) An attribute of a new process after it is created by a currently active process. *ISO.1.*

partitioned concatenation. Specifying multiple PDSs or PDSEs under one ddname. The concatenated data sets act as one big PDS or PDSE and access can be made to any member with a unique name. An attempted access to a member whose name occurs more than once in the concatenated data sets, returns the first member with that name found in the entire concatenation.

partitioned data set (PDS). A data set in direct access storage that is divided into partitions, called members, each of which can contain a program, part of a program, or data. *IBM.*

partitioned data set extended (PDSE). Similar to *partitioned data set*, but with extended capabilities.

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 treated 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 are treated as a single slash. The interpretation of the path name is described in *path name resolution*. *ISO.1.* (2) A file name specifying all directories leading to the file.

path name resolution. Path name resolution is performed for a process to resolve a path name to a particular file in a file hierarchy. There may be multiple path names that resolve to the same file. *X/Open.*

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.*

PCH (precompiled header). One or more headers that have already been compiled.

period. The character (.). The term *period* is contrasted against *dot*, which is used to describe a specific directory entry. This character is named <period> in the portable character set.

permissions. Codes that determine how a file can be used by any users who work on the system. See also *file access permissions*. *IBM*.

persistent environment. A program can explicitly establish a persistent environment, direct functions to it, and explicitly terminate it.

pointer. In the C and C++ languages, a variable that holds the address of a data object or a function. *IBM*.

pointer class. A class that implements pointers.

pointer to member. An operator used to access the address of non-static members of a class.

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>	0
<asterisk>	1
<plus-sign>	2
<comma>	3
<hyphen>	4
<hyphen-minus>	5
<period>	6
<slash>	7
<zero>	8
<one>	9
<two>	:
<three>	;
<four>	<
<five>	=
<six>	>
<seven>	?
<eight>	@
<nine>	
<colon>	
<semicolon>	
<less-than-sign>	
<equals-sign>	
<greater-than-sign>	
<question-mark>	
<commercial-at>	

<A>	A
	B
<C>	C
<D>	D
<E>	E
<F>	F
<G>	G
<H>	H
<I>	I
<J>	J
<K>	K
<L>	L
<M>	M
<N>	N
<O>	O
<P>	P
<Q>	Q
<R>	R
<S>	S
<T>	T
<U>	U
<V>	V
<W>	W
<X>	X
<Y>	Y
<Z>	Z
<left-square-bracket>	[
<backslash>	\
<reverse-solidus>	\
<right-square-bracket>]
<circumflex>	^
<circumflex-accent>	^
<underscore>	_
<low-line>	~
<grave-accent>	`
<a>	a
	b
<c>	c
<d>	d
<e>	e
<f>	f
<g>	g
<h>	h
<i>	i
<j>	j
<k>	k
<l>	l
<m>	m
<n>	n
<o>	o
<p>	p
<q>	q
<r>	r
<s>	s
<t>	t
<u>	u
<v>	v
<w>	w
<x>	x
<y>	y
<z>	z

```

<left-brace>      {
<left-curly-bracket> {
<vertical-line>  |
<right-brace>    }
<right-curly-bracket> }
<tilde>          ~

```

portable file name character set. The set of characters from which portable file names are constructed. For a file name to be portable across implementations conforming to the ISO POSIX-1 standard and to ISO/IEC 9945, it must consist only of the following characters:

```

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
0 1 2 3 4 5 6 7 8 9 . _ -

```

The last three characters are the period, underscore, and hyphen characters, respectively. The hyphen must not be used as the first character of a portable file name. Upper- and lower-case letters retain their unique identities between conforming implementations. In the case of a portable path name, the slash character may also be used. *X/Open. ISO.1.*

portability. The ability of a programming language to compile successfully on different operating systems without requiring changes to the source code.

positional parameter. A parameter that must appear in a specified location relative to other positional parameters. *IBM.*

precedence. The priority system for grouping different types of operators with their operands.

precompiled header. See *PCH*.

predefined macros. Frequently used routines provided by an application or language for the programmer.

preinitialization. A process by which an environment or library is initialized once and can then be used repeatedly to avoid the inefficiency of initializing the environment or library each time it is needed.

prelinker. A utility provided with OS/390 Language Environment that you can use to process application programs that require DLL support, or contain either constructed reentrancy or external symbol names that are longer than 8 characters. You require the prelinker, or its equivalent function which is provided by the binder, to process all C++ applications, or C applications that are compiled with the RENT, DLL, LONGNAME or IPA options. As of Version 2 Release 4, the prelinker was superseded by the binder. See also *binder*.

preprocessor. A phase of the compiler that examines the source program for preprocessor statements that

are then executed, resulting in the alteration of the source program.

preprocessor statement. In the C and C++ languages, a statement that begins with the symbol # and is interpreted by the preprocessor during compilation. *IBM.*

primary expression. (1) An identifier, parenthesized expression, function call, array element specification, structure member specification, or union member specification. *IBM.* (2) Literals, names, and names qualified by the :: (scope resolution) operator.

printable character. One of the characters included in the print character classification of the LC_CTYPE category in the current locale. *X/Open. ISO.1.*

private. Pertaining to a class member that is only accessible to member functions and friends of that class.

process. (1) An instance of an executing application and the resources it uses. (2) An address space and single thread of control that executes within that address space, and its required system resources. A process is created by another process issuing the fork() function. The process that issues the fork() function is known as the parent process, and the new process created by the fork() function is known as the child process. *X/Open. ISO.1.*

process group. A collection of processes that permits the signaling of related processes. Each process in the system is a member of a process group that is identified by the process group ID. A newly created process joins the process group of its creator. *IBM. X/Open. ISO.1.*

process group ID. The unique identifier representing a process group during its lifetime. A process group ID is a positive integer. (Under ISO only, it is a positive integer *that can be contained in a pid_t.*) A process group ID will not be reused by the system until the process group lifetime ends. *X/Open. ISO.1.*

process group lifetime. A period of time that begins when a process group is created and ends when the last remaining process in the group leaves the group, because either it is the end of the last process' lifetime or the last remaining process is calling the setsid() or setpgid() functions. *X/Open. ISO.1.*

process ID. The unique identifier representing a process. A process ID is a positive integer. (Under ISO only, it is a positive integer *that can be contained in a pid_t.*) A process ID will not be reused by the system until the process lifetime ends. In addition, if there exists a process group whose process group ID is equal to that process ID, the process ID will not be reused by the system until the process group lifetime ends. A

process that is not a system process will not have a process ID of 1. *X/Open. ISO.1.*

process lifetime. The period of time that begins when a process is created and ends when the process ID is returned to the system. After a process is created with a fork() function, it is considered active. Its thread of control and address space exist until it terminates. It then enters an inactive state where certain resources may be returned to the system, although some resources, such as the process ID, are still in use. When another process executes a wait() or waitpid() function for an inactive process, the remaining resources are returned to the system. The last resource to be returned to the system is the process ID. At this time, the lifetime of the process ends. *X/Open. ISO.1.*

program object. All or part of a computer program in a form suitable for loading into main storage for execution. A program object is the output of the OS/390 Binder and is a newer more flexible format (e.g. longer external names) than a load module.

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 parameters. See *function prototype*.

public. Pertaining to a class member that is accessible to all functions.

pure virtual function. A virtual function that has a function definition of = 0;. See also *abstract classes*.

Q

qualified class name. Any class name or class name qualified with one or more :: (scope resolution) operators.

qualified name. Used to qualify a nonclass type name such as a member by its class name.

qualified type name. Used to reduce complex class name syntax by using typedefs to represent qualified class names.

Query Management Facility (QMF). Pertaining to an IBM query and report writing facility that enables a variety of tasks such as data entry, query building, administration, and report analysis. *IBM.*

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.

quotation marks. The characters " and ', also known as *double-quote* and *single-quote* respectively. *X/Open*.

R

radix character. The character that separates the integer part of a number from the fractional part. *X/Open*.

real group ID. The attribute of a process that, at the time of process creating, identifies the group of the user who created the process. This value is subject to change during the process lifetime, as describe in `setgid()`. *X/Open*. *ISO.1*.

real user ID. The attribute of a process that, at the time of process creation, identifies the user who created the process. This value is subject to change during the process lifetime, as described in `setuid()`. *X/Open*. *ISO.1*.

reason code. A code that identifies the reason for a detected error. *IBM*.

reassociation. An optimization technique that rearranges the sequence of calculations in a subscript expression producing more candidates for common expression elimination.

redirection. In the shell, a method of associating files with the input or output of commands. *X/Open*.

reentrant. The attribute of a program or routine that allows the same copy of a program or routine to be used concurrently by two or more tasks.

reference class. A class that links a concrete class to an abstract class. Reference classes make polymorphism possible with the Collection Classes. Synonymous with *indirection class*.

refresh. To ensure that the information on the user's terminal screen is up-to-date. *X/Open*.

register storage class specifier. A specifier that indicates to the compiler within a block scope data definition, or a parameter declaration, that the object being described will be heavily used.

register variable. A variable defined with the register storage class specifier. Register variables have automatic storage.

regular expression. (1) A mechanism to select specific strings from a set of character strings. (2) A set of characters, meta-characters, and operators that define a string or group of strings in a search pattern.

(3) A string containing wildcard characters and operations that define a set of one or more possible strings.

regular file. A file that is a randomly accessible sequence of bytes, with no further structure imposed by the system. *X/Open*. *ISO.1*.

relation. An unordered flat collection class that uses keys, allows for duplicate elements, and has element equality.

relative path name. The name of a directory or file expressed as a sequence of directories followed by a file name, beginning from the current directory. See *path name resolution*. *IBM*.

reserved word. (1) In programming languages, a keyword that may not be used as an identifier. *ISO-JTC1*. (2) A word used in a source program to describe an action to be taken by the program or compiler. It must not appear in the program as a user-defined name or a system name. *IBM*.

RMODE (residency mode). In MVS, a program attribute that refers to where a module is prepared to run. RMODE can be 24 or ANY. ANY refers to the fact that the module can be loaded either above or below the 16M line. RMODE 24 means the module expects to be loaded below the 16M line.

runtime library. A compiled collection of functions whose members can be referred to by an application program during runtime execution. Typically used to refer to a dynamic library that is provided in object code, such that references to the library are resolved during the linking step. The runtime library itself is not statically bound into the application modules.

S

saved set-group-ID. An attribute of a process that allows some flexibility in the assignment of the effective group ID attribute, as described in the `exec()` family of functions and `setgid()`. *X/Open*. *ISO.1*.

saved set-user-ID. An attribute of a process that allows some flexibility in the assignment of the effective user ID attribute, as described in `exec()` and `setuid()`. *X/Open*. *ISO.1*.

scalar. An arithmetic object, or a pointer to an object of any type.

scope. (1) That part of a source program in which a variable is visible. (2) 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. Synonymous with *scope resolution operator*.

scope resolution operator (::). Synonym for *scope operator*.

semaphore. An object used by multi-threaded applications for signalling purposes and for controlling access to serially reusable resources. Processes can be locked to a resource with semaphores if the processes follow certain programming conventions.

sequence. A sequentially ordered flat collection.

sequential concatenation. Multiple sequential data sets or partitioned data-set members are treated as one long sequential data set. In the case of sequential data sets, you can access or update the data sets in order. In the case of partitioned data-set members, you can access or update the members in order. Repositioning is possible if all of the data sets in the concatenation support repositioning.

sequential data set. A data set whose records are organized on the basis of their successive physical positions, such as on magnetic tape. *IBM*.

session. A collection of process groups established for job control purposes. Each process group is a member of a session. A process is a member of the session of which its process group is a member. A newly created process joins the session of its creator. A process can alter its session membership; see `setsid()`. There can be multiple process groups in the same session. *X/Open. ISO.1*.

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 the OS/390 Shell and Utilities feature licensed program.

Short name. An external non-C++ name in an object module produced by compiling with the `NOLONGNAME` option. Such a name is up to 8 characters long and single case.

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) A method of interprocess communication that simulates software interrupts. *IBM*.

signal handler. A function to be called when the signal is reported.

single-byte character set (SBCS). A set of characters in which each character is represented by a one-byte code. *IBM*.

single-precision. Pertaining to the use of one computer word to represent a number in accordance with the required precision. *ISO-JTC1. ANSI/ISO*.

single-quote. The character `'`, also known as *apostrophe*. This character is named `<quotation-mark>` in the portable character set.

slash. The character `/`, also known as *solidus*. This character is named `<slash>` in the portable character set.

socket. (1) A unique host identifier created by the concatenation of a port identifier with a transmission control protocol/Internet protocol (TCP/IP) address. (2) A port identifier. (3) A 16-bit port-identifier. (4) A port on a specific host; a communications end point that is accessible through a protocol family's addressing mechanism. A socket is identified by a socket address. *IBM*.

sorted map. A sorted flat collection with key and element equality.

sorted relation. A sorted flat collection that uses keys, has element equality, and allows duplicate elements.

sorted set. A sorted flat collection with element equality.

source module. 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*.

spanned record. A logical record contained in more than one block. *IBM*.

specialization. A user-supplied definition which replaces a corresponding template instantiation.

specifiers. Used in declarations to indicate storage class, fundamental data type and other properties of the object or function being declared.

spill area. A storage area used to save the contents of registers. *IBM.*

SQL (Structured Query Language). A language designed to create, access, update and free data tables.

square brackets. The characters [(left bracket) and] (right bracket). Also see *brackets*.

stack frame. The physical representation of the activation of a routine. The stack frame is allocated and freed on a LIFO (last in, first out) basis. A stack is a collection of one or more stack segments consisting of an initial stack segment and zero or more increments.

stack storage. Synonym for *automatic storage*.

standard error. An output stream usually intended to be used for diagnostic messages. *X/Open.*

standard input. (1) An input stream usually intended to be used for primary data input. *X/Open.* (2) The primary source of data entered into a command. 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. *IBM.*

standard output. (1) An output stream usually intended to be used for primary data output. *X/Open.* (2) The primary destination of data coming from a command. Standard output goes to the display unless redirection or piping is used, in which case standard output can go to a file or to another command. *IBM.*

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.

static binding. The act of resolving references to external variables and functions before run time.

storage class specifier. One of the terms used to specify a storage class, such as auto, register, static, or extern.

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. Such objects can be created by the `fopen()` or `fopen()` functions, and are associated with a file descriptor. A stream provides the additional services of user-selectable buffering and formatted input and output. *X/Open.*

string. A contiguous sequence of bytes terminated by and including the first null byte. *X/Open.*

string constant. Zero or more characters enclosed in double quotation marks.

string literal. Zero or more characters enclosed in double quotation marks.

striped data set. A special data set organization that spreads a data set over a specified number of volumes so that I/O parallelism can be exploited. Record n in a striped data set is found on a volume separate from the volume containing record $n - p$, where $n > p$.

struct. An aggregate of elements having arbitrary types.

structure. A construct (a class data type) that contains an ordered group of data objects. Unlike an array, the data objects within a structure can have varied data types. A structure can be used in all places a class is used. The initial projection is public.

structure tag. The identifier that names a structure data type.

Structured Query Language. See *SQL*.

stub routine. A routine, within a runtime library, that contains the minimum lines of code required to locate a given routine at run time.

subprogram. In the IPA Link version of the Inline Report listing section, an equivalent term for 'function'.

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.

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.

support. In system development, to provide the necessary resources for the correct operation of a functional unit. *IBM.*

switch expression. The controlling expression of a switch statement.

switch statement. A C or C++ language statement that causes control to be transferred to one of several statements depending on the value of an expression.

system default. A default value defined in the system profile. *IBM.*

System Object Model (SOM). Defines an IBM interface between programs, or between libraries and programs, so that an object's interface is separated from its implementation. SOM allows classes of objects to be defined in one programming language and used in another, and it allows libraries of such classes to be updated without requiring client code to be recompiled. *IBM.*

system process. (1) An implementation-dependent object, other than a process executing an application, that has a process ID. *X/Open.* (2) An object, other than a process executing an application, that is defined by the system, and has a process ID. *ISO.1.*

T

tab character. A character that in the output stream indicates that printing or displaying should start at the next horizontal tabulation position on the current line. The tab is the character designated by '\t' in the C language. If the current position is at or past the last defined horizontal tabulation position, the behavior is unspecified. It is unspecified whether the character is the exact sequence transmitted to an output device by the system to accomplish the tabulation. *X/Open.*

This character is named <tab> in the portable character set.

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 with variable types.

template class. A class instance generated by a class template.

Template Declaration. A prototype of a template which can optionally include a template definition.

Template Definition. A blueprint the compiler uses to generate a template instantiation.

template function. A function generated by a function template.

Template Instantiation. Compiler generated code for a class or function using the referenced types and the corresponding class or function template definition.

terminals. Synonym for *leaves.*

text file. A file that contains characters organized into one or more lines. The lines must not contain NUL 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.*

thread. The smallest unit of operation to be performed within a process. *IBM.*

throw expression. An argument to the C++ exception being thrown.

tilde. The character ~. This character is named <tilde> in the portable character set.

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. *IBM.*

traceback. A section of a dump that provides information about the stack frame, the program unit address, the entry point of the routine, the statement number, and the status of the routines on the call-chain at the time the traceback was produced.

trigraph sequence. An alternative spelling of some characters to allow the implementation of C in character sets that do not provide a sufficient number of non-alphabetic graphics. *ANSI/ISO.*

Before preprocessing, each trigraph sequence in a string or literal is replaced by the single character that it represents.

truncate. To shorten a value to a specified length.

try block. A block in which a known C++ exception is passed to a handler.

type conversion. Synonym for *boundary alignment.*

type definition. A definition of a name for a data type. *IBM.*

type specifier. Used to indicate the data type of an object or function being declared.

U

ultimate consumer. The target of data in an I/O operation. An ultimate consumer can be a file, a device, or an array of bytes in memory.

ultimate producer. The source of data in an I/O operation. An ultimate producer can be a file, a device, or an array of bytes in memory.

unary expression. An expression that contains one operand. *IBM.*

undefined behavior. Action by the compiler and library when the program uses erroneous constructs or contains erroneous data. Permissible undefined behavior includes ignoring the situation completely with unpredictable results. It also includes behaving in a documented manner that is characteristic of the environment, during translation or program execution, with or without issuing a diagnostic message. It can also include terminating a translation or execution, while issuing a diagnostic message. Contrast with *unspecified behavior* and *implementation-defined behavior*.

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) In the C or C++ language, a variable that can hold any one of several data types, but only one data type at a time. *IBM.* (2) For bags, there is an additional rule for duplicates: If bag P contains an element m times and bag Q contains the same element n times, then the union of P and Q contains that element $m+n$ times.

union tag. The identifier that names a union data type.

unnamed pipe. A pipe that is accessible only by the process that created the pipe and its child processes. An unnamed pipe does not have to be opened before it can be used. It is a temporary file that lasts only until the last file descriptor that uses it is closed.

unique collection. A collection in which the value of an element only occurs once; that is, there are no duplicate elements.

unrecoverable error. An error for which recovery is impossible without use of recovery techniques external to the computer program or run.

unspecified behavior. Action by the compiler and library when the program uses correct constructs or data, for which the standards impose no specific requirements. Such action should not cause compiler or application failure. You should not, however, write any programs to rely on such behavior as they may not

be portable to other systems. Contrast with *implementation-defined behavior* and *undefined behavior*.

user-defined data type. (1) 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. (2) See also *abstract data type*.

user ID. A nonnegative integer that is used to identify a system user. (Under ISO only, a nonnegative integer, which can be contained in an object of type *uid_t*.) When the identity of a user is associated with a process, a user ID value is referred to as a real user ID, an effective user ID, or (under ISO only, and there optionally) a saved set-user ID. *X/Open. ISO.1.*

user name. A string that is used to identify a user. *ISO.1.*

user prefix. In an MVS environment, the user prefix is typically the user's logon user identification.

V

value numbering. An optimization technique that involves local constant propagation, local expression elimination, and folding several instructions into a single instruction.

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.*

variant character. A character whose hexadecimal value differs between different character sets. On EBCDIC systems, such as S/390, these 13 characters are an exception to the portability of the portable character set.

<left-square-bracket>	[
<right-square-bracket>]
<left-brace>	{
<right-brace>	}
<backslash>	\
<circumflex>	^
<tilde>	~
<exclamation-mark>	!
<number-sign>	#
<vertical-line>	
<grave-accent>	`
<dollar-sign>	\$
<commercial-at>	@

vertical-tab character. A character that in the output stream indicates that printing should start at the next vertical tabulation position. The vertical-tab is the character designated by '\v' in the C or C++ languages. If the current position is at or past the last

defined vertical tabulation position, the behavior is unspecified. It is unspecified whether this character is the exact sequence transmitted to an output device by the system to accomplish the tabulation. *X/Open*. This character is named <vertical-tab> in the portable character set.

virtual address space. (1) In virtual storage systems, the virtual storage assigned to a batched or terminal job, a system task, or a task initiated by a command. (2) In VSE, a subdivision of the virtual address area available to the user for the allocation of private, non-shared partitions.

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, which is determined at run time.

Virtual Storage Access Method (VSAM). An access method for direct or sequential processing of fixed and variable length records on direct access devices. The records in a VSAM data set or file can be organized in logical sequence by a key field (key sequence), in the physical sequence in which they are written on the data set or file (entry-sequence), or by relative-record number.

visible. Visibility of identifiers is based on scoping rules and is independent of *access*.

volatile attribute. (1) In the C or C++ language, the keyword *volatile*, used in a definition, declaration, or cast. It causes the compiler to place the value of the data object in storage and to reload this value at each reference to the data object. *IBM*. (2) An attribute of a data object that indicates the object is changeable. Any expression referring to a volatile object is evaluated immediately (for example, assignments).

W

while statement. A looping statement that contains the keyword *while* followed by an expression in parentheses (the condition) and a statement (the action). *IBM*.

white space. (1) Space characters, tab characters, form-feed characters, and new-line characters. (2) A sequence of one or more characters that belong to the space character class as defined via the `LC_CTYPE` category in the current locale. In the POSIX locale, white space consists of one or more blank characters (space and tab characters), new-line characters, carriage-return characters, form-feed characters, and vertical-tab characters. *X/Open*.

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.

wide-character code. An integral value corresponding to a single graphic symbol or control code. *X/Open*.

wide-character string. A contiguous sequence of wide-character codes terminated by and including the first null wide-character code. *X/Open*.

wide-oriented stream. See *orientation of a stream*.

working directory. Synonym for *current working directory*.

writable static area. See *WSA*.

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*. *ANSI/ISO*.

WSA (writable static area). An area of memory in the program that is modifyable during program execution. Typically, this area contains global variables and function and variable descriptors for DLLs.

Bibliography

This bibliography lists the publications for IBM products that are related to the OS/390 C/C++ product. It includes publications covering the application programming task. The bibliography is not a comprehensive list of the publications for these products, however, it should be adequate for most OS/390 C/C++ users. Refer to the *OS/390 Information Roadmap*, GC28-1727, for a complete list of publications belonging to the OS/390 product.

Related publications not listed in this section can be found on the IBM Online Library Omnibus Edition: MVS Collection CD-ROM (SK2T-0710), the *IBM Online Library Omnibus Edition: OS/390 Collection* CD-ROM (SK2T-6700), or on a tape available with OS/390.

OS/390

- *OS/390 Printing Softcopy BOOKs*, S544-5354
- *OS/390 Introduction and Release Guide*, GC28-1725
- *OS/390 Planning for Installation*, GC28-1726
- *OS/390 Summary of Message Changes*, GC28-1499
- *OS/390 Information Roadmap*, GC28-1727

VS COBOL II Release 4

- *General Information*, GC26-4042
- *Migration Guide for MVS and CMS*, GC26-3151
- *Installation and Customization for MVS*, SC26-4048
- *Application Programming Guide for MVS and CMS*, SC26-4045
- *Application Programming Language Reference*, GC26-4047
- *Application Programming Reference Summary*, SX26-3721
- *Application Programming Debugging*, SC26-4049
- *Application Programming Diagnosis Guide*, LY27-9523
- *Application Programming Diagnosis Reference*, LY27-9522

COBOL FOR MVS & VM Release 2

- *Compiler and Run-Time Migration Guide*, GC26-4764
- *Programming Guide*, SC26-4767
- *Language Reference*, SC26-4769
- *Diagnosis Guide*, SC26-3138
- *Licensed Program Specifications*, GC26-4761
- *Installation and Customization under MVS*, SC26-4766

COBOL for OS/390 & VM Version 2 Release 1

- *Compiler and Run-Time Migration Guide*, GC26-4764
- *Programming Guide*, SC26-9049
- *Language Reference*, SC26-9046
- *Diagnosis Guide*, GC26-9047
- *Licensed Program Specifications*, GC26-9044
- *Installation and Customization under OS/390*, GC26-9045
- *Program Directory for VM*
- *Fact Sheet*, GC26-9048

PL/I for MVS & VM Release 1 Modification 1

- *Language Reference*, SC26-3114
- *Compiler and Run-Time Migration Guide*, SC26-3118
- *Programming Guide*, SC26-3113
- *Compile-Time Messages and Codes*, SC26-3229
- *Reference Summary*, SX26-3821
- *Diagnosis Guide*, SC26-3149
- *Installation and Customization under MVS*, SC26-3119
- *Licensed Program Specifications*, GC26-3116

OS PL/I Version 2 Release 3

- *Programming Guide*, SC26-4307
- *Programming: Language Reference*, SC26-4308
- *Programming: Messages and Codes*, SC26-4309

VS FORTRAN Version 2 Release 6

- *Programming Reference*, SC26-4221
- *Programming Guide*, SC26-4222

CICS/ESA Version 4 Release 1

- *Application Programming Reference*, SC33-1170
- *Application Programming Guide*, SC33-1169
- *Installation Guide*, SC33-1163
- *System Definition Guide*, SC33-1164
- *Resource Definition Guide*, SC33-1166
- *Messages and Codes*, SC33-1177

CICS Transaction Server for OS/390 Release 2

- *Application Programming Guide*, SC33-1687
- *Application Programming Reference*, SC33-1688
- *System Programming Reference*, SC33-1689
- *Distributed Transaction Programming Guide*, SC33-1691
- *Front End Programming Interface User's Guide*, SC33-1692

DB2 Version 3 Release 1

- *SQL Reference*, SC26-4890
- *Reference Summary*, SX26-3801
- *Command and Utility Reference*, SC26-4891
- *Application Programming and SQL Guide*, SC26-4889

DB2 Version 4 Release 1

- *SQL Reference*, SC26-3270
- *Reference Summary*, SX26-3829
- *Command Reference*, SC26-3267
- *Application Programming and SQL Guide*, SC26-3266
- *Utility Guide and Reference*, SC26-3395

DB2 Version 5 Release 1

- *Administration Guide*, SC26-8957
- *Application Programming and SQL Guide*, SC26-8958
- *Call Level Interface Guide and Reference*, SC26-8959
- *Command Reference*, SC26-8960
- *Data Sharing: Planning and Administration*, SC26-8961
- *Installation Guide*, GC26-8970
- *Messages and Codes*, GC26-8979
- *SQL Reference*, SC26-8966
- *Reference for Remote DRDA Requesters and Servers*, SC26-8964
- *Utility Guide and Reference*, SC26-8967

IMS/ESA Version 4 Release 1

- *Application Programming: Design Guide*, SC26-3066
- *Application Programming: DL/I Calls*, SC26-3062
- *Application Programming: Data Communication*, SC26-3058
- *Application Programming: EXEC DL/I Commands*, SC26-3063

IMS/ESA Version 5 Release 1

- *Application Programming: Design Guide*, SC26-8016
- *Application Programming: Transaction Manager*, SC26-8017
- *Application Programming: Database Manager*, SC26-8015
- *Application Programming: EXEC DL/I Commands for CICS and IMS*, SC26-8018

IMS/ESA Version 6 Release 1

- *Application Programming: Design Guide*, SC26-8728
- *Application Programming: Transaction Manager*, SC26-8729
- *Application Programming: Database Manager*, SC26-8727
- *Application Programming: EXEC DL/I Commands for CICS and IMS*, SC26-8726

QMF Version 3 Release 2

- *Introducing QMF*, GC26-4713
- *Using QMF*, SC26-8078
- *Developing QMF Applications*, SC26-4722
- *Reference*, SC26-4716
- *Managing QMF for MVS*, SC26-8218
- *Reference*, SC26-4716

- *Messages and Codes*, SC26-4834
- *Installing on MVS*, SC26-4719

VSAM

- *MVS/ESA VSAM Catalog Administration: Access Method Services Reference*, SC26-4501
- *MVS/ESA VSAM Administration: Macro Instruction Reference*, SC26-4517
- *MVS/ESA VSAM Administration Guide for MVS/DFP*, SC26-4518
- *MVS/ESA Integrated Catalog Administration: Access Method Services Reference*, SC26-4500
- *DFSMS/MVS Access Method Services for VSAM*, SC26-4905
- *MVS/DFP Access Method Services for VSAM Catalogs*, SC26-4570
- *MVS/Extended Architecture VSAM Catalog Administration: Access Method Services Reference (Data Facility Product, Version 2)*, GC26-4136

Index

Special Characters

- ~Cursor
 - IObserverList::Cursor 495
 - IThread::Cursor 556
- ~IApplication
 - IApplication 424
- ~ICurrentApplication
 - ICurrentApplication 445
- ~INotificationEvent
 - INotificationEvent 480
- ~INotifier
 - INotifier 484
- ~IObserver
 - IObserver 487
- ~IObserverList
 - IObserverList 492
- ~IPrivateResource
 - IPrivateResource 500
- ~IRefCounted
 - IRefCounted 509
- ~IReference
 - IReference 512
- ~IResource
 - IResource 515
- ~IResourceLock
 - IResourceLock 520
- ~ISharedResource
 - ISharedResource 524
- ~IStandardNotifier
 - IStandardNotifier 530
- ~IThread
 - IThread 539
- ~IThreadFn
 - IThreadFn 560
- ~IThreadMemberFn
 - IThreadMemberFn 566
- ~PrivateSemaphoreHandle
 - IPrivateSemaphoreHandle 503

A

- abs() complex function 8
- abstract Collection Classes
 - collection 283
 - equality collection 285
 - equality key collection 287
 - equality key sorted collection 289
 - equality sorted collection 291
 - key collection 293
 - key sorted collection 295
 - ordered collection 297

- abstract Collection Classes (*continued*)
 - sequential collection 299
 - sorted collection 301
- IAccessError class 349
- add
 - IObserverList 492
- add() Collection Class function 98
- addAllFrom() Collection Class function 99
- addAsChild() tree function 245
- addAsFirst() Collection Class function 100
- addAsLast() Collection Class function 100
- addAsNext() Collection Class function 101
- addAsPrevious() Collection Class function 101
- addAsRoot() tree function 245
- addAtPosition() Collection Class function 101
- addDifference() Collection Class function 102
- addIntersection() Collection Class function 103
- addLocation() IException function 343
- addObserver
 - INotifier 485
 - IStandardNotifier 532
- addOrReplaceElementWithKey() Collection Class function 103
- addRef
 - IRefCounted 508
- addRef() IBuffer function 315
- addUnion() Collection Class function 104
- adjustArg() I0String function 411
- adjustPriority
 - IApplication 422
 - IThread 550
- adjustResult() I0String function 411
- allElementsDo() function
 - flat collections 105
 - tree collections 245—246
- allocate() function
 - IBuffer class 324
 - IDBCSBuffer class 326
 - streambuf class 77
- allSubtreeElementsDo() tree function 245—246
- anchorBlock
 - ICurrentThread 450
- any() Collection Class function 106
- appContext
 - ICurrentThread 451
- appendText() IException function 344
- IApplication 421
- Application Support Class Library
 - I0String class 405
 - IAccessError class 349
 - IAssertionFailure class 350
 - IBase class 311

Application Support Class Library *(continued)*

- IBaseErrorInfo class 354
- IBuffer class 315
- ICLibErrorInfo class 351
- IDate class 335
- IDBCSBuffer class 325
- IDeviceError class 353
- IException class 341
- IException::TraceFn class 357
- IExceptionLocation class 358
- IGUIErrorInfo class 359
- IInvalidParameter class 362
- IInvalidRequest class 363
- IMessageText class 364
- IOutOfMemory class 366
- IOutOfSystemResource class 367
- IOutOfWindowResource class 368
- IResourceExhausted class 369
- IString class 377
- IStringEnum class 412
- IStringParser class 413
- IStringParser::SkipWords class 416
- IStringTest class 417
- IStringTestMemberFn class 418
- ISystemErrorInfo class 371
- ITime class 569
- ITrace class 579
- IVBase class 313
- IXLibErrorInfo class 373
- VBase class 313
- applicator class 265
- applyBitOp() IString function 401
- appShell
 - ICurrentThread 451
- arg() complex function 9
- arg1, arg2 c_exception arguments 11
- argc
 - ICurrentApplication 443
- argv
 - ICurrentApplication 444
- array initialization in complex class 4
- array stream buffer classes 83—86
- asBCD
 - Decimal 441
- asCDATE() IDate function 336
- asCTIME() ITime function 570
- asDebugInfo
 - IApplication 422
 - ICurrentApplication 444
 - IEnumHandle 457
 - IHandle 472
 - IPrivateSemaphoreHandle 504
 - IProcessId 505
 - ISharedSemaphoreHandle 527
 - IThread 539
 - IThreadHandle 562

asDebugInfo *(continued)*

- IThreadId 564
- asDebugInfo() function
 - IBase class 311
 - IBuffer class 315
 - IString class 379
 - IVBase class 314
- asDouble
 - IBinaryCodedDecimal 435
- asDouble() IString function 379
- asInt() IString function 379
- asLong
 - IBinaryCodedDecimal 435
 - IEventData 462
- asLongLong
 - IBinaryCodedDecimal 435
- asLongLong() IString function 380
- asSeconds
 - ITimeStamp 577
- asSeconds() ITime function 570
- IAssertionFailure class 350
- assertParameter() IException function 344
- assignment operator
 - See operator =
- asString
 - Decimal 441
 - IApplication 422
 - IBinaryCodedDecimal 435
 - IEnumHandle 458
 - IHandle 472
 - IPrivateSemaphoreHandle 504
 - IProcessId 506
 - ISharedSemaphoreHandle 528
 - IThread 540
 - IThreadHandle 562
 - IThreadId 564
 - ITimeStamp 576
- asString() function
 - IBase class 311
 - IDate class 336
 - IString class 380
 - ITime class 570
 - IVBase class 314
- asUnsigned
 - IEnumHandle 458
 - IHandle 472
 - IPrivateSemaphoreHandle 504
 - IProcessId 506
 - ISharedSemaphoreHandle 528
 - IThreadHandle 562
 - IThreadId 564
- asUnsigned() IString function 380
- asUnsignedLong
 - IEventData 462
- asUnsignedLongLong() IString function 380

- AT&T C++ Language System Release 1.2
 - overflow() 79
 - setbuf() 81
 - streambuf constructor 73
 - underflow() 81
- attach() filebuf function 18
- attach() fstreambase function 24
- attachAsChild() tree function 247
- attachAsRoot() tree function 247
- attachSubtreeAsChild() tree function 247
- attachSubtreeAsRoot() tree function 247
- autoInitGUI
 - IThread 540

B

- b2c() IString function 380
- b2d() IString function 380
- b2x() IString function 380
- bad() ios function 39
- bag 125—129
- IBase class 311
- base() streambuf function 75
- based-on concept in Collection Class Library
 - functions affected by
 - addAsFirst() 100
 - addAsLast() 100
 - addAsNext() 101
 - addAsPrevious() 101
 - addAtPostion() 102
 - addOrReplaceElementWithKey() 103
 - enqueue() 109
 - isBounded() 110
 - isFull() 111
 - locateOrAddElementWithKey() 114
 - maxNumberOfElements() 115
 - push() 117
 - general precondition 99
- IBaseErrorInfo class
- baseLibrary
 - Exception 347
- IBinaryCodedDecimal 427, 436
- bitalloc() ios function 38
- blen() streambuf function 77
- IBoolean 93
- BooleanConstants IBase enumeration 313
- IBuffer class 315
- buffer() IString function 401
- built-in manipulators
 - istream class 52
 - ostream class 64

C

- c_exception class 11—13

- c2b() IString function 380
- c2d() IString function 381
- c2x() IString function 381
- cData
 - Decimal 440
 - IBinaryCodedDecimal 434
- center() function
 - IBuffer class 315
 - IDBCSBuffer class 326
 - IString class 381
- change() function
 - IOString class 407
 - IBuffer class 315
 - IString class 381, 401
- char1
 - EventData 461
- char2
 - EventData 461
- char3
 - EventData 461
- char4
 - EventData 461
- character conversion for numeric input 48
- charLength() IDBCSBuffer function 331
- CharType enumeration of IStringEnum class 412
- charType() function
 - IOString class 408
 - IBuffer class 316
 - IDBCSBuffer class 326
 - IString class 382
- checkAddition() IBuffer function 316
- checkMultiplication() IBuffer function 316
- childPositionAt() tree function 248
- children of a tree node 237
- CLASS_BASE_NAME 93
- CLASS_NAME 93
- className() function
 - IBuffer class 324
 - IDBCSBuffer class 331
- clear() ios function 39
- clearLock
 - IResourceLock 520
- ICLibErrorInfo class 351
- CLibrary
 - Exception 347
- close() function
 - filebuf class 18
 - fstreambase class 24
- collection
 - conditions for equality 98
- collection abstract class 283
- Collection Class Library
 - abstract classes
 - collection 283
 - equality collection 285
 - equality key collection 287
 - equality key sorted collection 289

Collection Class Library (*continued*)

- abstract classes (*continued*)
 - equality sorted collection 291
 - key collection 293
 - key sorted collection 295
 - ordered collection 297
 - sequential collection 299
 - sorted collection 301
- applicator classes 265
- concrete classes
 - bag collection 125
 - deque collection 131
 - equality sequence collection 137
 - heap collection 141
 - key bag collection 145
 - key set collection 151
 - key sorted bag collection 157
 - key sorted set collection 163
 - map collection 171
 - priority queue collection 179
 - queue collection 183
 - relation collection 187
 - sequence collection 191
 - set collection 197
 - sorted bag collection 203
 - sorted map collection 207
 - sorted relation collection 219
 - sorted set collection 223
 - stack collection 229
- cursor classes 257
- pointer classes 267
- tree cursor classes 261
- Collection Event Data 273
- Collection Guard 275
- Command IStringParser enumeration 416
- compare() function
 - IBuffer class 316
 - Collection Class Library 106
- complex class 3—9
 - constants 3
 - conversion functions 8
 - error handling 11
 - input operator 6
 - mathematical operators 5
 - output operator 6
- Complex Mathematics Library 3—13
- complex_error() Complex Mathematics function 12
- conj() complex function 9
- conjugates of complex numbers 9
- constant applicator class 265
- constructors
 - IAccessError class 349
 - IAssertionFailure class 350
 - IBuffer class 315
 - ICLibErrorInfo class 351
 - Collection Class Library
 - flat collections 97

constructors (*continued*)

- complex class 4
- cursor classes 258
- IDate class 335
- IDBCSBuffer class 326
- IDeviceError class 353
- IException class 343
- IExceptionLocation class 358
- filebuf class 18
- flat collections 97
- fstream class 24
- IGUIErrorInfo class 360
- ifstream class 26
- IInvalidParameter class 362
- IInvalidRequest class 363
- iostream class 43
- iostream_withassign class 43
- istream class 45
- istream_withassign class 53
- istrstream class 84
- IMessageText class 364
- ofstream class 28
- ostream class 59
- ostream_withassign class 65
- ostrstream class 85
- IOutOfMemory class 367
- IOutOfSystemResource class 368
- IOutOfWindowResource class 369
- IResourceExhausted class 370
- stdiobuf class 67
- stdiostream class 68
- IString class 378, 406
- IStringParser class 414
- IStringParser::SkipWords class 416
- IStringTest class 417
- IStringTestMemberFn class 419
- strstream class 84
- strstreambuf class 87
- ISystemErrorInfo class 372
- ITime class 569
- ITrace class 580
- tree class 244
- tree cursor class 261
- IXLibErrorInfo class 374
- contains() Collection Class function 106
- containsAllFrom() Collection Class function 106
- containsAllKeysFrom() Collection Class function 107
- containsElementWithKey() Collection Class function 107
- contents() IBuffer function 316
- conversion functions in complex class 8
- copy constructors
 - flat collections 97
 - tree class 244
- copy() function
 - IBuffer class 316

- copy() function (*continued*)
 - flat collections 107
 - IString class 382
 - tree class 248
- copySubtree() tree function 248
- cos() complex function 8
- cosh() complex function 8
- current
 - IApplication 423
 - IThread 548
- ICurrentApplication 443
- currentHandle
 - IThread 549
- currentId
 - IThread 549
- currentPID
 - IApplication 424
- ICurrentThread 447
- currentTimeStamp
 - ITimeStamp 575
- cursor
 - IObserverList::Cursor 495
 - IThread::Cursor 556
 - IVCollectionEventData 273
- cursor classes 257—259

D

- d2b() IString function 382
- d2c() IString function 382
- d2x() IString function 383
- data() IString function 402
- IDate class 335
- dayName() IDate function 337
- dayOfMonth() IDate function 337
- DayOfWeek IDate enumeration 340
- dayOfWeek() IDate function 337
- dayOfYear() IDate function 337
- daysInMonth() IDate function 337
- daysInYear() IDate function 337
- IDBCSBuffer class 325
- dbp() streambuf function 77
- ios::dec 34, 42
- Decimal
 - Decimal 436
- defaultAutoInitGUI
 - IThread 540
- defaultBuffer() function
 - IBuffer class 316
 - IString class 402
- defaultQueueSize
 - IThread 542
- defaultStackSize
 - IThread 544
- degree of a tree node 237

- deleteId
 - IStandardNotifier 533
- deque 131—135
- deque() Collection Class function 108
- Destination ITrace enumeration 583
- destructors
 - complex class 4
 - filebuf class 18
 - flat collections 97
 - istrstream class 85
 - ostrstream class 85
 - stdiobuf class 68
 - IString class 396
 - strstream class 84
 - strstreambuf class 88
 - tree class 244
- detach() function
 - filebuf class 18
 - fstreampbase class 24
- IDeviceError class 353
- dialogControls
 - IThread 541
- difference
 - definition for bags 125
 - definition for flat collections 108
- differenceWith() Collection Class function 108
- digitsOf
 - Decimal 440
 - IBinaryCodedDecimal 434
- disableInternationalization() IString function 383
- disableNotification
 - INotifier 484
 - IStandardNotifier 530
- disableNotification() Collection Class function 108
- disableTrace() ITrace function 581
- disableWriteLineNumber() ITrace function 581
- disableWritePrefix() ITrace function 581
- dispatchNotificationEvent
 - IObserver 489
- doallocate() function
 - streambuf class 79
 - strstreambuf class 88
- dynamic mode 84, 87

E

- e (mathematical constant) 3
- eback() streambuf function 75
- ebuf() streambuf function 75
- egptr() streambuf function 75
- element
 - IVCollectionEventData 273
- element() function
 - cursor classes 258
 - tree cursor class 262

- elementAt
 - IObservableList 492
- elementAt() function
 - flat collection classes 108—109
 - tree class 248
- elementAtPosition() Collection Class function 109
- elementWithKey() Collection Class function 109
- enableInternationalization() IString function 383
- enableNotification
 - INotifier 484
 - IStandardNotifier 530
- enableNotification() Collection Class function 109
- enableTrace() ITrace function 581
- enableWriteLineNumber() ITrace function 581
- enableWritePrefix() ITrace function 581
- endl manipulator 64
- ends manipulator 64
- enqueue() Collection Class function 109
- EnumHandle 457
- eof() ios function 39
- epptr() streambuf function 75
- equal element 93
- equality collection abstract class 285
- equality key collection abstract class 287
- equality key sorted collection abstract class 289
- equality sequence 137—140
- equality sorted collection abstract class 291
- error
 - handling
 - for complex class 11—13
 - ios error state 39
 - messages
 - Complex Mathematics Library 12
- errorCodeGroup
 - IException 343
- errorId() function
 - IBaseErrorInfo class 355
 - ICLibErrorInfo class 352
 - IException class 344
 - IGUIErrorInfo class 360
 - ISystemErrorInfo class 372
 - IXLibErrorInfo class 374
- ErrorInfo class 354
- IEventData 459
 - INotificationEvent 480
- IEventParameter1 465
- IEventParameter2 467
- IEventResult 469
- examples
 - bag 128
 - heap 167
 - key bag 147
 - key set 155
 - key sorted bag 160
 - key sorted set 167
 - machine-readable lix

- examples (*continued*)
 - map 176
 - naming of lix
 - sequence 194
 - set 201
 - softcopy lix
 - sorted map 211
 - sorted relation 211
 - sorted set 226
- IException class 341
- IExceptionLocation class 358
- exceptionLogged
 - IException::TraceFn 348
- IException::TraceFn class 357
- ExceptionType IBaseErrorInfo enumeration 356
- exit
 - ICurrentApplication 444
 - ICurrentThread 448
- exp() complex function 7

F

- fail() ios function 40
- fd() filebuf function 19
- file attributes
 - in filebuf open() function 19
 - in fstream constructor 25
 - in fstream open() function 25
 - in ifstream constructor 27
 - in ifstream open() function 27
 - in ofstream constructor 28
 - in ofstream open() function 28
- filebuf class 17—21
- filebuf::openprot 27
- fileName() IExceptionLocation function 358
- fill() ios function 36
- findPhrase() IString function 402
- first() Collection Class function 110
- ios::fixed 35
- flags() ios function 36
- flush manipulator 64
- flush() ostream function 64
- FnType IStringTest enumeration 418
- fopen() library function 68
- format flags in ios class
 - mutually exclusive flags 35
 - predefined 33—36
 - user-defined 38
- format state
 - fill character 36
 - flags 33—36
 - introduction 32
 - parameterized manipulators 55
 - precision 57
 - width variable 46, 47, 60

- format variables 32
- formatting
 - of input streams 46
 - of output streams 60
- fp() filebuf function 19
- freeze() function
 - ostream class 86
 - strstream class 84
 - strstreambuf class 88
- fromContents() IBuffer function 317
- fstream class 24—26
- fstreambase class 23—24
- functionName() IExceptionLocation function 359

G

- gbump() streambuf function 78
- gcount() istream function 51
- get() istream function 49, 50
- getline() istream function 50
- good() ios function 40
- gptr() streambuf function 75
- IGUIErrorInfo class 359

H

- handle 471
 - ICurrentThread 448
 - IHandle 473
 - IThread 549
- handleNotificationsFor
 - IObserver 488
- hasChild() tree function 249
- hasNotifierAttrChanged
 - INotificationEvent 480
- header files
 - See chapters on individual classes
- heap 141—143
 - example of 167
- height of a tree 237
- ios::hex 34, 42
- IHighEventParameter 475
- highHighByte
 - IEventData 461
- highLowByte
 - IEventData 461
- highNumber
 - IEventData 461
- hours() ITime function 570

I

Note: Most classes beginning with an uppercase ‘I’ are listed under their second letter

- I/O Stream Library
 - filebuf class 17

I/O Stream Library (*continued*)

- fstream class 24
- fstreambase class 23
- ifstream class 26
- ios class 31
- iostream class 43
- iostream_withassign class 43
- istream class 45
- istream_withassign class 53
- istrstream class 84
- ofstream class 28
- ostream class 59
- ostream_withassign class 65
- ostrstream class 85
- parameterized manipulators 55
- stdiobuf class 67
- stdiostream class 68
- streambuf class 71
- strstream class 84
- strstreambase class 83
- strstreambuf class 87
- IApplication
 - IApplication 424
- IBinaryCodedDecimal
 - IBinaryCodedDecimal 427
- ICurrentApplication
 - ICurrentApplication 445
- ICurrentThread
 - ICurrentThread 455
- id
 - IApplication 424
 - ICurrentThread 448
 - IThread 549
- IEnumHandle
 - IEnumHandle 457
- IEventData
 - IEventData 460
- ifstream class 26—28
- ignore() istream function 50
- IHandle
 - IHandle 472
- imag() complex function 9
- imaginary part of a complex number 4
- implementation variant
 - See chapters on individual Collection Classes
- in_avail() streambuf function 73
- includes() IString function 383
- includesDBCS() function
 - IBuffer class 317
 - IDBCSBuffer class 326
 - IString class 383
- includesMBCS() function
 - IBuffer class 317
 - IDBCSBuffer class 326
 - IString class 383

- includesSBCS() function
 - IBuffer class 317
 - IDBCSBuffer class 326
 - IString class 383
- indexOf() function
 - IString class 408
 - IBuffer class 317
 - IDBCSBuffer class 327
 - IString class 383
- indexOfAnyBut() function
 - IString class 408
 - IBuffer class 317
 - IDBCSBuffer class 327
 - IString class 384
- indexOfAnyOf() function
 - IString class 408
 - IBuffer class 317
 - IDBCSBuffer class 327
 - IString class 384
- indexOfPhrase() function
 - IString class 408
 - IString class 384
- indexOfWord() function
 - IString class 408
 - IString class 384, 402
- initBuffer() IString function 402
- initialize() function
 - IBuffer class 324
 - IDate class 340
 - ITime class 572
- initializeGUI
 - ICurrentThread 451
- INotificationEvent
 - INotificationEvent 479
- Notifier
 - Notifier 484
- input operator
 - See operator >>
- insert() function
 - IString class 408
 - IBuffer class 318
 - IDBCSBuffer class 327
 - IString class 384, 403
- ios::internal 34
- internal classes
 - fstreambase class 23
 - strstreambase class 83
- intersection
 - bags 125
 - flat collections 110
- intersectionWith() Collection Class function 110
- invalidate
 - IObserverList::Cursor 496
 - IThread::Cursor 556
- invalidate() function
 - cursor classes 258
- invalidate() function (*continued*)
 - tree cursor class 262
- InvalidParameter class 362
- InvalidRequest class 363
- IObserver
 - IObserver 488
- IObserverList
 - IObserverList 491
- ios class 31—42
 - built-in manipulators 42
 - error checking 39
 - error state 39
 - format state
 - base conversion 34
 - buffer flushing 35
 - floating-point formatting 34
 - integral formatting 34
 - member functions 36
 - uppercase and lowercase 35
 - white space and padding 33
 - format state variables 32
- ios::app 25
- ios::ate 25
- ios::beg 64, 89
- ios::cur 64, 89
- ios::dec 34, 48, 62
- ios::end 64, 89
- ios::failbit 27, 29, 48
- ios::fixed 35
- ios::hex 34, 48, 62
- ios::in 26, 27, 89
- ios::internal 34
- ios::left 33
- ios::nocreate 26, 28
- ios::noreplace 26
- ios::oct 34, 48, 62
- ios::out 26, 89
- ios::right 34
- ios::scientific 34
- ios::showbase 34
- ios::showpoint 34
- ios::showpos 34
- ios::skipws 33
 - preventing looping 33
- ios::stdio 35, 67
- ios::trunc 26
- ios::unitbuf 35
- ios::uppercase 35
- ios::x_fill 56
- ios::x_prec 57
- ios::x_width 46, 47, 60
- iostream class 43—44
- istream_withassign class 43—44
- ipfx() istream function 45
- IPrivateResource
 - IPrivateResource 500

- IPrivateSemaphoreHandle
 - IPrivateSemaphoreHandle 503
- IProcessId
 - IProcessId 505
- IRefCounted
 - IRefCounted 508
- IReference
 - IReference 512
- IResource
 - IResource 515
- IResourceLock
 - IResourceLock 519
- is_open() filebuf function 19
- isAbbrevFor() IString function 403
- isAbbreviationFor() IString function 385
- isAlphabetic() function
 - IBuffer class 318
 - IString class 385
- isAlphanumeric() function
 - IBuffer class 318
 - IString class 385
- isASCII() function
 - IBuffer class 318
 - IString class 385
- isAvailable() function
 - IBaseErrorInfo class 355
 - ICLibErrorInfo class 352
 - IGUIErrorInfo class 361
 - ISystemErrorInfo class 372
 - IXLibErrorInfo class 374
- isBinaryDigits() IString function 385
- isBounded() Collection Class function 110
- isCharValid() IDBCSBuffer function 331
- isControl() function
 - IBuffer class 318
 - IString class 386
- isDBCS() function
 - IBuffer class 318
 - IDBCSBuffer class 327
 - IString class 386
- isDBCS1() IDBCSBuffer function 332
- isDigits() function
 - IBuffer class 318
 - IString class 386
- isEmpty
 - IObserverList 492
- isEmpty() function
 - flat collections 110
 - tree class 249
- isEnabledForNotification
 - INotifier 484
 - IStandardNotifier 531
- isEnabledForNotification() Collection Class function 111
- isFirstAt() Collection Class function 111
- isFull() Collection Class function 111
- isGraphics() function
 - IBuffer class 318
 - IString class 386
- isGUIInitialized
 - ICurrentThread 452
- ISharedResource
 - ISharedResource 524
- ISharedSemaphoreHandle
 - ISharedSemaphoreHandle 527
- isHexDigits() function
 - IBuffer class 319
 - IString class 386
- isInternationalized() IString function 386
- isLastAt() Collection Class function 111
- isLeaf() tree function 249
- isLeapYear() IDate function 337
- isLike() IString function 386, 403
- isLowerCase() function
 - IBuffer class 319
 - IString class 387
- isMBCS() function
 - IBuffer class 319
 - IDBCSBuffer class 327
 - IString class 387
- isNegative
 - Decimal 441
 - IBinaryCodedDecimal 434
- isPositive
 - Decimal 441
 - IBinaryCodedDecimal 434
- isPrevDBCS() IDBCSBuffer function 332
- isPrintable() function
 - IBuffer class 319
 - IString class 387
- isPunctuation() function
 - IBuffer class 319
 - IString class 387
- isRecoverable() IException function 344
- isRoot() tree function 249
- isSBC() IDBCSBuffer function 332
- isSBCS() function
 - IBuffer class 319
 - IDBCSBuffer class 327
 - IString class 387
- isStarted
 - IThread 549
- IStandardNotifier
 - IStandardNotifier 530
- isTopLevelShell
 - ICurrentThread 448
- isTraceEnabled() ITrace function 581
- istream class 45—53
 - assignment operator 45
 - built-in manipulators 52
 - formatted input 46

- istream class (*continued*)
 - input operator 47—49
 - input prefix function 45
 - unformatted input 49
- istream_withassign class 45—53
- IString... classes
 - See entries for IString... under S
- istrstream class 83—86
- isUpperCase() function
 - IBuffer class 319
 - IString class 388
- isValid
 - IObserverList::Cursor 496
 - IThread::Cursor 556
 - IThreadId 564
- isValid() function
 - cursor classes 258
 - tree cursor class 262
- isValid() IDate function 337
- isValidDBCS() function
 - IBuffer class 320
 - IDBCSBuffer class 327
 - IString class 388
- isValidMBCS() function
 - IBuffer class 320
 - IDBCSBuffer class 328
 - IString class 388
- isWhiteSpace() function
 - IBuffer class 320
 - IString class 388
- isWriteLineNumberEnabled() ITrace function 581
- isWritePrefixEnabled() ITrace function 582
- isXErrorCodeAvailable
 - ICurrentThread 449
- iteration order 93, 240
- IThread
 - IThread 537
- IThreadFn
 - IThreadFn 559
- IThreadHandle
 - IThreadHandle 561
- IThreadId
 - IThreadId 563
- IThreadMemberFn
 - IThreadMemberFn 566
- ITimeStamp
 - ITimeStamp 575
- ITreeCursor::invalidate 262
- ITreeCursor::isValid 262
- ITreeCursor::setToChild 262
- ITreeCursor::setToFirstExistingChild 262
- ITreeCursor::setToLastExistingChild 262
- ITreeCursor::setToNextExistingChild 263
- ITreeCursor::setToParent 263
- ITreeCursor::setToPreviousExistingChild 263

- ITreeCursor::setToRoot 263
- IVBase class 313
- iword() ios function 38

J

- julianDate() IDate function 338

K

- key bag 145—150
- key collection abstract class 293
- key set 151—156
- key sorted bag 157—161
- key sorted collection abstract class 295
- key sorted set 163—170
- key() Collection Class function 111
- keyName
 - ISharedResource 525

L

- last-in, first-out behavior (LIFO) 229
- last() Collection Class function 111
- lastIndexOf() function
 - IString class 409
 - IBuffer class 320
 - IDBCSBuffer class 328
 - IString class 388
- lastIndexOfAnyBut() function
 - IString class 409
 - IBuffer class 320
 - IDBCSBuffer class 328
 - IString class 389
- lastIndexOfAnyOf() function
 - IString class 409
 - IBuffer class 321
 - IDBCSBuffer class 328
 - IString class 389
- leaves of a tree 237
- ios::left 33
- leftJustify() function
 - IBuffer class 321
 - IDBCSBuffer class 328
 - IString class 389
- length() function
 - IBuffer class 321
 - IString class 389
- lengthOf() IString function 404
- lengthOfWord() IString function 390
- level of a tree node 237
- LIFO (last-in, first-out) behavior 229
- lineFrom() IString function 390
- lineNumber() IExceptionLocation function 359
- linked sequence 193

- locate() Collection Class function 112
- locateElementWithKey() Collection Class function 112
- locateFirst() Collection Class function 112
- locateLast() Collection Class function 112
- locateNext() Collection Class function 113
- locateNextElementWithKey() Collection Class function 113
- locateOrAdd() Collection Class function 113
- locateOrAddElementWithKey() Collection Class function 114
- locatePrevious() Collection Class function 115
- locationAtIndex() IException function 344
- locationCount() IException function 345
- lock
 - IResource 516
- log() complex function 7
- logExceptionData() IException function 345
- lowerCase() function
 - IBuffer class 321
 - IDBCSBuffer class 329
 - IString class 390
- ILowEventParameter 477
- lowHighByte
 - IEventData 461
- lowLowByte
 - IEventData 462
- lowNumber
 - IEventData 462
- lrecl parameter
 - in filebuf open() function 19
 - in fstream constructor 25
 - in fstream open() function 25
 - in ifstream constructor 27
 - in ifstream open() function 27
 - in ofstream constructor 28
 - in ofstream open() function 28

M

- map 171—178
- mathematical constants defined by complex class 3
- mathematical functions for complex class 7
- maxCharLength() IDBCSBuffer function 332
- maxNumberOfElements() Collection Class function 115
- messageFile() IBase function 312
- messageQueue
 - ICurrentThread 452
 - IThread 542
- IMessageText class 364
- messageText() IBase function 312
- minutes() ITime function 570
- Month IDate enumeration 340
- monthName() IDate function 338
- monthOfYear() IDate function 338

- multiway tree class 239—244

N

- n-ary tree class
 - cursor class for 261
- name data member of c_exception class 11
- name() function
 - IAccessError class 349
 - IAssertionFailure class 350
 - IDeviceError class 353
 - IException class 345
 - IInvalidParameter class 363
 - IInvalidRequest class 364
 - IOutOfMemory class 367
 - IOutOfSystemResource class 368
 - IOutOfWindowResource class 369
 - IResourceExhausted class 370
- newBuffer() IBuffer function 321
- newCursor() function
 - flat collections 115
 - tree class 249
- newStartedThread
 - IThread 552
- next() function
 - IBuffer class 322
 - IDBCSBuffer class 329
- node of a tree 237
- noHandle
 - IThreadHandle 562
- norm() complex function 8
- INotificationEvent 479
- notificationId
 - INotificationEvent 480
- notifier 483
 - INotificationEvent 481
- notifier() Collection Class function 115
- notifyObservers
 - INotifier 485, 486
 - IObserverList 493
 - IStandardNotifier 531, 532
- notifyObservers() Collection Class function 116
- now() ITime function 570
- null() IBuffer function 322
- INumber Collection Class type 93
- number1
 - IEventData 462
- number2
 - IEventData 462
- numberOfChildren() tree function 250
- numberOfDifferentElements() Collection Class function 116
- numberOfDifferentKeys() Collection Class function 116
- numberOfElements
 - IObserverList 492

- numberOfElements() function
 - flat collections 116
 - tree class 250
- numberOfElementsWithKey() Collection Class
 - function 116
- numberOfLeaves() tree function 250
- numberOfOccurrences() Collection Class function 116
- numberOfSubtreeElements() tree function 250
- numberOfSubtreeLeaves() tree function 250
- numberOfWords() function
 - IStringParser::SkipWords class 417
- numWords() function
 - IString class 390

O

- IObserver 487
- observerData
 - INotificationEvent 481
- observerList 491
 - INotifier 486
 - IStandardNotifier 532
- IObserverList::Cursor 495
- obsolete declarations
 - overflow() 79
 - streambuf constructor 73
- obsolete versions of functions
 - setbuf() 81
 - underflow() 81
- occurrencesOf() function
 - I0String class 410
 - IString class 390, 404
- ios::oct 34, 42
- ofstream class 28—29
- open_mode enumeration 25
- open() function
 - filebuf class 19
 - file attribute parameter 19
 - fstream class 25
 - ifstream class 27
 - ofstream class 28
- operatingSystem
 - IException 347
- operator ~
 - See destructors
- operator -
 - complex class 5
 - IDate class 338
 - Decimal 439
 - IBinaryCodedDecimal 432
 - ITimeStamp 576
 - ITime class 571
- operator --
 - Decimal 439
 - IBinaryCodedDecimal 432
- operator -=
 - complex class 6
 - IDate class 339
 - Decimal 439
 - IBinaryCodedDecimal 432
 - ITimeStamp 577
 - ITime class 571
- operator ->
 - IReference 513
- operator !
 - Decimal 438
 - IBinaryCodedDecimal 431
 - ios class 40
- operator !=
 - complex class 6
 - cursor classes 258
 - IDate class 338
 - Decimal 438
 - flat collections 97
 - IBinaryCodedDecimal 430
 - IString class 390
 - ITimeStamp 574
 - ITime class 571
- operator /
 - complex class 5
 - Decimal 439
- operator /=
 - complex class 6
 - Decimal 440
 - IBinaryCodedDecimal 433
- operator []
 - I0String class 410
 - IString class 395
- operator ()
 - ios class 40
- operator *
 - complex class 5
 - Decimal 439
 - IReference 513
- operator *=
 - complex class 6
 - Decimal 439
 - IBinaryCodedDecimal 431
- operator &
 - IString class 391
- operator &=
 - IString class 391
- operator +
 - complex class 5
 - IDate class 338
 - Decimal 439
 - IBinaryCodedDecimal 431
 - ITimeStamp 576
 - IString class 391
 - ITime class 571

operator ++
 Decimal 439
 IBinaryCodedDecimal 431
 operator +=
 complex class 6
 IDate class 339
 Decimal 439
 IBinaryCodedDecimal 432
 IString class 392
 ITimeStamp 576
 ITime class 571
 operator <
 IDate class 339
 Decimal 438
 IBinaryCodedDecimal 430
 IString class 392
 ITimeStamp 574
 ITime class 571
 operator <<
 IBase class 312
 char values 61
 complex class 6
 IDate class 339
 float and double values 62
 integral values 62
 ostream class 60—63
 pointers to void 63
 stream buffers 63
 IString class 392
 IStringParser class 415
 ITime class 571
 IVBase class 314
 operator <=
 IDate class 339
 Decimal 438
 IBinaryCodedDecimal 431
 IString class 392
 ITimeStamp 574
 ITime class 571
 operator =
 Decimal 440
 flat collections 98
 IBinaryCodedDecimal 433
 INotificationEvent 480
 ios class 32
 iostream class 43
 IPrivateSemaphoreHandle 504
 IReference 512
 IStandardNotifier 530
 istream_withassign class 53
 IThread 552
 IThreadId 564
 IMessageText class 366
 ostream_withassign class 65
 IString class 393
 tree class 244
 operator ==
 complex class 5
 cursor classes 258
 IDate class 339
 Decimal 438
 flat collections 98
 IBinaryCodedDecimal 431
 IString class 393
 ITimeStamp 574
 ITime class 571
 operator >
 IDate class 339
 Decimal 438
 IBinaryCodedDecimal 431
 IString class 393
 ITimeStamp 574
 ITime class 571
 operator >=
 IDate class 339
 Decimal 438
 IBinaryCodedDecimal 431
 IString class 394
 ITimeStamp 574
 ITime class 572
 operator >>
 arrays of char 47
 complex class 6
 float and double values 48
 integral values 47
 istream class 46—49
 pointers to char 47
 references to char 47
 streambuf objects 49
 IString class 394
 IStringParser class 415
 operator ^
 IString class 395
 operator |
 IString class 395
 operator |=
 IString class 396
 operator char *
 IEventData 462
 IString class 394
 operator const char *
 IBaseErrorInfo class 355
 ICLibErrorInfo class 352
 IGUIErrorInfo class 361
 IMessageText class 366
 ISystemErrorInfo class 372
 IXLibErrorInfo class 375
 operator const void*
 ios class 40
 operator delete
 IBuffer class 324

- operator IDate
 - ITimeStamp 577
- operator ITime
 - ITimeStamp 577
- operator new
 - IBuffer class 324
- operator signed char *
 - IString class 394
- operator T *
 - IReference 513
- operator unsigned char *
 - IString class 394
- operator unsigned long
 - IEventData 462
- operator Value
 - IEnumHandle 458
 - IHandle 472
 - IPrivateSemaphoreHandle 504
 - IProcessId 506
 - ISharedSemaphoreHandle 528
- operator void*
 - ios class 40
- operator!=
 - tree cursor class 261
- operator==
 - tree cursor class 261
- opfx() ostream function 60
- ordered collection abstract class 297
- osfx() ostream function 60, 67
- ostream class 59—65
 - built-in manipulators 64
 - formatted output 60
 - output operator 61
 - output prefix function 60
 - output suffix function 60
 - positioning 64
 - unformatted output 63
- ostream_withassign class 65
- ostrstream class 83—86
- other
 - Exception 347
- out_waiting() streambuf function 73
- IOOutOfMemory class 366
- IOOutOfSystemResource class 367
- IOOutOfWindowResource class 368
- output operator
 - See operator <<
- output suffix function 60
- overflow errors in complex class 8
- overflow() function
 - IBuffer class 322
 - streambuf class 79
 - strstreambuf class 89
- overlayWith() function
 - IString class 410
 - IBuffer class 322

- overlayWith() function (*continued*)
 - IDBCSBuffer class 329
 - IString class 396, 404

P

- parameterized manipulators
 - format state 55
 - introduction 55
 - resetiosflags() 56
 - setbase() 56
 - setfill() 56
 - setiosflags() 56
 - setprecision() 57
 - setw() 57
- parent in a tree 237
- pbackfail() streambuf function 79
- pbase() streambuf function 75
- pbump() streambuf function 78
- pcount() ostrstream function 86
- peek() istream function 51
- pi (mathematical constant) 3
- pib
 - ICurrentApplication 446
- pointer class 267—269
- polar() complex function 9
- pop() Collection Class function 116
- IPosition Collection Class type 93
- positionAt() Collection Class function 117
- positioning property 93
- IPostorder Collection Class type 93
- postorder traversal of trees 237—238
- pow() complex function 7
- pptr() streambuf function 76
- precision() ios function 36
- precisionOf
 - Decimal 441
 - IBinaryCodedDecimal 435
- IPreorder Collection Class type 93
- preorder traversal of trees 237—238
- presentationSystem
 - Exception 347
- prevCharLength() IDBCSBuffer function 332
- priority queue 179—182
 - deque() 108
 - enqueue() 109
- priorityClass
 - IThread 550
- priorityLevel
 - IThread 551
- IPrivateResource 499
- IPrivateSemaphoreHandle 503
- IProcessId 505
- processMsgs
 - ICurrentThread 452

pthread_t
 IThreadId 564
 push() Collection Class function 117
 put() ostream function 63
 putback() istream function 52
 pword() ios function 38

Q

queue collection 183—186
 deque() 108
 enqueue() 109
 queueSize
 IThread 543

R

rdbuf() function
 fstream class 26
 ifstream class 27
 ios class 41
 ofstream class 29
 stdiostream class 68
 strstreambase class 83
 rdstate() ios function 40
 read() istream function 50
 readFromStream
 IBinaryCodedDecimal 435
 real part of a complex number 4
 real() complex function 9
 recfm parameter
 in filebuf open() function 19
 in fstream constructor 25
 in fstream open() function 25
 in ifstream constructor 27
 in ifstream open() function 27
 in ofstream constructor 28
 in ofstream open() function 28
 IRefCounted 507
 IReference 511
 reference class
 See chapters on individual Collection Classes
 relatedHandlesList
 IThread 541
 relation 187—189
 remainingStack
 ICurrentThread 449
 remove
 IObserverList 492
 remove() function
 IString class 410
 IBuffer class 322
 Collection Class Library 117
 IDBCSBuffer class 329
 IString class 396

removeAll
 IObserverList 493
 removeAll() function
 flat collections 118
 tree class 250
 removeAllElementsWithKey() Collection Class
 function 118
 removeAllObservers
 INotifier 486
 IStandardNotifier 532
 removeAllOccurrences() Collection Class function 118
 removeAt
 IObserverList 493
 removeAt() Collection Class function 119
 removeAtPosition() Collection Class function 119
 removeElementWithKey() Collection Class
 function 119
 removeFirst() Collection Class function 120
 removeLast() Collection Class function 120
 removeObserver
 INotifier 486
 IStandardNotifier 532
 removeRef
 IRefCounted 508
 removeRef() IBuffer function 322
 removeSubtree() tree function 250
 removeWords() IString function 397
 replaceAt() function
 flat collections 120
 tree class 251
 replaceElementWithKey() Collection Class
 function 121
 repositioning the get or put pointers 89
 resetiosflags() manipulator 56
 IResource 515
 IResourceExhausted class 369
 IResourceLock 519
 Restricted Access Collection 303
 Restricted Access Collection Guard 277
 resume
 IThread 546
 returned element 240
 retval 12
 reverse 121
 reverse() function
 IBuffer class 323
 IDBCSBuffer class 329
 IString class 397
 ios::right 34
 rightJustify() function
 IBuffer class 323
 IDBCSBuffer class 329
 IString class 397
 root of a tree 237
 run
 ICurrentApplication 445

run (*continued*)
 IThreadFn 560
 IThreadMemberFn 566

S

same key 93
 sbumpc() streambuf function 73
 ios::scientific 34
 seconds() ITime function 572
 secondsInDay
 ITimeStamp 578
 seekg() function
 istream class 51
 ostream class 85
 seekoff() function
 filebuf class 19
 streambuf class 80
 strstreambuf class 89
 seekp() ostream function 64
 seekpos() function
 filebuf class 20
 streambuf class 80
 sequence 191—196
 sequence as table 193
 sequential collection abstract class 299
 sequential collections
 add() behavior with 98
 conditions for equality 98
 set collection 197—202
 setArgs
 ICurrentApplication 444
 setAutoInitGUI
 IThread 540
 setb() streambuf function 76
 setbase() manipulator 56
 setbuf() function
 filebuf class 20
 fstreambase class 24
 streambuf class 81
 strstreambuf class 90
 setBuffer() IString function 404
 setDefaultAutoInitGUI
 IThread 541
 setDefaultBuffer() IBuffer function 323
 setDefaultQueueSize
 IThread 543
 setDefaultStackSize
 IThread 544
 setDefaultText() IMessageText function 366
 setErrorCodeGroup
 IException 343
 setErrorId() IException function 345
 setEventData
 INotificationEvent 481

setf() ios function 37
 setfill() manipulator 56
 setg() streambuf function 76
 setid
 IApplication 425
 setiosflags() manipulator 56
 setLock
 IResourceLock 521
 setMessageFile() IBase function 313
 setNotifierAttrChanged
 INotificationEvent 481
 setObserverData
 INotificationEvent 481
 setp() streambuf function 76
 setprecision() manipulator 57
 setPriority
 IApplication 423
 IThread 551
 setQueueSize
 IThread 543
 setRelatedHandlesList
 IThread 541
 setSeverity() IException function 345
 setStackSize
 IThread 545
 setText() IException function 345
 setToChild() function
 tree class 251
 tree cursor class 262
 setToFirst
 IObserverList::Cursor 496
 IThread::Cursor 556
 setToFirst() function
 cursor classes 259
 flat collections 121
 tree class 251
 setToFirstExistingChild() function
 tree class 251
 tree cursor class 262
 setToLast
 IObserverList::Cursor 496
 setToLast() function
 cursor classes 259
 flat collections 121
 tree class 252
 setToLastExistingChild() function
 tree class 252
 tree cursor class 262
 setToNext
 IObserverList::Cursor 496
 IThread::Cursor 556
 setToNext() function
 cursor classes 259
 flat collections 122
 tree class 252

- setToNextDifferentElement() Collection Class function 122
- setToNextExistingChild() function
 - tree class 252
 - tree cursor class 263
- setToNextWithDifferentKey() Collection Class function 122
- setToParent() function
 - tree class 253
 - tree cursor class 263
- setTopLevelShell
 - ICurrentThread 453
- setPosition() Collection Class function 123
- setToPrevious
 - IObserverList::Cursor 496
- setToPrevious() function
 - cursor classes 259
 - flat collections 123
 - tree class 253
- setToPreviousExistingChild() function
 - tree class 253
 - tree cursor class 263
- setToRoot() function
 - tree class 253
 - tree cursor class 263
- setTraceFunction() IException function 345
- setVariable
 - IThread 549
- setw() manipulator 57
- setWindowList
 - IThread 542
- setXErrorCode
 - ICurrentThread 453
- Severity enumeration of IException class 346
- sgetc() streambuf function 73
- sgetn() streambuf function 74
- ISharedResource 523
- ISharedSemaphoreHandle 527
- ios::showbase 34
- ios::showpoint 34
- ios::showpos 34
- sibling of a tree node 237
- sin() complex function 8
- sinh() complex function 8
- size() IString function 397
- skip() ios function 37
- ios::skipws 33
- sleep
 - ICurrentThread 449
- snxctc() streambuf function 74
- sort() Collection Class function 123
- sorted bag 203—206
- sorted collection abstract class 301
- sorted collections
 - add() behavior with 98
- sorted map 207—217
- sorted relation 219—222
 - example of 211
- sorted set 223—228
- space() IString function 397
- sputbackc() streambuf function 74
- sputc() streambuf function 74
- sputn() streambuf function 74
- sqrt() complex function 7
- square root of a complex number 7
- stack 229—234
- stackSize
 - IThread 545
- IStandardNotifier 529
- start
 - IThread 546
- startBackwardsSearch() function
 - IBuffer class 325
 - IDBCSBuffer class 333
- startedThread
 - ICurrentThread 455
 - IThread 552
- startSearch() function
 - IBuffer class 325
 - IDBCSBuffer class 333
- stderr 67
- ios::stdio 35
- stdiobuf class 67—68
- stdiofile() stdiobuf function 68
- stdiostream class 68—69
- stdout 67
- stop
 - IThread 548
- stopHandlingNotificationsFor
 - IObserver 488
- stopProcessingMsgs
 - IThread 541
- stossc() streambuf function 74
- str() function
 - ostrstream class 86
 - strstream class 84
 - strstreambuf class 89
- stream buffer boundaries 75
- streambuf class 71—82
- IString class 377—405
- IStringEnum class 412
- IStringParser class 413
- IStringParser::SkipWords class 416
- IStringTest class 417
- IStringTestMemberFn class 418
- strip() function
 - IBuffer class 323
 - IDBCSBuffer class 329
 - IString class 397, 404
- stripBlanks() IString function 398

- stripLeading() IString function 398
- stripLeadingBlanks() IString function 398
- StripMode IStringEnum enumeration 413
- stripTrailing() IString function 398
- stripTrailingBlanks() IString function 399
- strstream class 83—86
- strstreambase class 83
- strstreambuf class 87—90
- subString() function
 - IString class 411
 - IBuffer class 323
 - IDBCSBuffer class 330
 - IString class 399
- subtree of a tree 237
- suspend
 - ICurrentThread 454
 - IThread 548
- sync_with_stdio() ios function 41
- sync() function
 - filebuf class 20
 - istream class 52
 - streambuf class 81
- ISystemErrorInfo class 371

T

- tellg() istream function 51
- tellp() ostream function 64
- template arguments
 - See chapters on individual Collection Classes
- terminal of a tree 237
- terminate() IException function 346
- terminateGUI
 - ICurrentThread 452
- test() function
 - IStringTest class 418
 - IStringTestMemberFn class 419
- text() function
 - IBaseErrorInfo class 355
 - ICLibErrorInfo class 352
 - IException class 346
 - IGUIErrorInfo class 361
 - IMessageText class 366
 - ISystemErrorInfo class 372
 - IXLibErrorInfo class 375
- textCount() IException function 346
- this collection 94
- this tree 240
- IThread 535
- IThread::Cursor 555
- IThreadFn 559
- IThreadHandle 561
- threadId 563
 - IThread::Cursor 556
- threadId() ITrace function 583

- IThreadMemberFn 565
- throwCLibError() ICLibErrorInfo function 352
- throwError() function
 - IBaseErrorInfo class 356
 - IGUIErrorInfo class 361
- throwGUIError() IGUIErrorInfo function 361
- throwSystemError() ISystemErrorInfo function 372
- throwXLibError() IXLibErrorInfo function 375
- tie() ios function 41
- ITime class 569
- ITimeStamp 573
- today() IDate function 339
- top() Collection Class function 124
- ITrace class 579
- traceDestination() ITrace function 582
- TraceFn
 - IException::TraceFn 348
- translate() function
 - IBuffer class 324
 - IDBCSBuffer class 330
 - IString class 399, 405
- tree 237—238
- tree class 239—244
- Tree Collection Guard 279
- tree cursor class 261
- ITreeIterationOrder Collection Class type 93
- trigonometric functions for complex class 8
- type member of c_exception class 12

U

- unbuffered() streambuf function 78
- undefined cursor 93
- underflow() function
 - streambuf class 81
 - strstreambuf class 90
- unformatted input 49
- unformatted output 63
- union
 - bags 125
 - flat collections 124
- unionWith() Collection Class function 124
- unique collections
 - add() behavior with 98
 - conditions for equality 98
- ios::unitbuf 35
- unlock
 - IResource 516
- unordered collections
 - locateNext() 113
 - locateNextElementWithKey() 113
- unsetf() ios function 37
- ios::uppercase 35
- upperCase() function
 - IBuffer class 324
 - IDBCSBuffer class 330

- upperCase() function (*continued*)
 - IString class 400
- useCount
 - IRefCounted 508
- useCount() IBuffer function 324
- user-defined format flags in ios class 38

V

- variable
 - IThread 550
- variant classes
 - See chapters on individual Collection Classes
- IVBase class 313
- version() IBase function 313

W

- waitFor
 - ICurrentThread 449
- waitForAllThreads
 - ICurrentThread 450
- waitForAnyThread
 - ICurrentThread 450
- width() ios function 37
- windowList
 - IThread 542
- word() IString function 400
- wordIndexOfPhrase() IString function 400
- words() IString function 400
- write() function
 - Exception::TraceFn class 357
 - ostream class 63
 - ITrace class 582
- writeFormattedString() ITrace function 583
- writeString() ITrace function 583
- writeToQueue() ITrace function 582
- writeToStandardError() ITrace function 582
- writeToStandardOutput() ITrace function 582
- writeToStream
 - IBinaryCodedDecimal 436

X

- x2b() IString function 400
- x2c() IString function 400
- x2d() IString function 401
- xalloc() ios function 39
- XerrorCode
 - ICurrentThread 454
- IXLibErrorInfo class 373

Y

- year() IDate function 339
- YearFormat IDate enumeration 340

Communicating Your Comments to IBM

OS/390
C/C++
IBM Open Class Library Reference

Publication No. SC09-2364-04

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@ca.ibm.com
 - IBMLink: [toribm\(torrcf\)](mailto:toribm(torrcf)@ca.ibm.com)
 - IBM/PROFS: [torolab4\(torrcf\)](mailto:torolab4(torrcf)@ca.ibm.com)
 - IBMMAIL: [ibmmail\(caibmwt9\)](mailto:ibmmail(caibmwt9)@ca.ibm.com)

Readers' Comments — We'd Like to Hear from You

OS/390

C/C++

IBM Open Class Library Reference

Publication No. SC09-2364-04

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.



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



SC09-2364-04

