

OS/390



C/C++ IBM Open Class Library User's Guide

OS/390



C/C++ IBM Open Class Library User's Guide

Note!

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

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	xi
Standards	xi
Trademarks	xii
 About This Book	 xv
IBM OS/390 C/C++ and Related Publications	xv
Hardcopy Books	xxii
Softcopy Books	xxii
Softcopy Examples	xxiii
OS/390 C/C++ on the World Wide Web	xxiii
C/C++ News...	xxiv
 About IBM OS/390 C/C++	 xxv
Changes for Version 2 Release 6	xxv
The C/C++ Compilers	xxvi
The C Language	xxvi
The C++ Language	xxvi
Common Features of the OS/390 C and C++ Compilers	xxvii
OS/390 C Compiler Specific Features	xxviii
Features That Are Specific to the OS/390 C++ Compiler	xxviii
Utilities	xxix
Class Libraries	xxix
Class Library Source	xxx
The Debug Tool	xxxi
OS/390 Language Environment	xxxi
The Program Management Binder	xxxii
OS/390 UNIX System Services (OS/390 UNIX)	xxxii
OS/390 C/C++ Applications with OS/390 UNIX C/C++ Functions	xxxiv
Input and Output	xxxv
I/O Interfaces	xxxv
File Types	xxxvi
Additional I/O Features	xxxvii
The System Programming C Facility	xxxvii
Interaction with Other IBM Products	xxxvii
Additional Features of OS/390 C/C++	xxxix
Suggested Reading	xl
 Chapter 1. Introduction to IBM Open Class Library	 1
History of IBM Open Class Library	1
Hierarchies of the Class Libraries	1
Coding with Class Libraries under OS/390 UNIX System Services	5
Compiling Programs that Use IBM Open Class Library	5
Binding with IBM Open Class Library	6
Migration Notes	6
Thread Safety and the IBM Open Class Library	7
Why Thread Safety?	7
Levels of Thread Safety	7

Part 1. Complex Mathematics Class Library	9
--	---

Chapter 2. Using the Complex Mathematics Classes	11
Review of Complex Numbers	11
Header Files and Constants for complex and c_exception	12
Constructing complex Objects	12
Complex Mathematics Input and Output	13
Mathematical Operators for complex	14
Equality and Inequality Operators Test for Absolute Equality	15
Assignment Operators Do Not Produce an lvalue	16
Friend Functions for complex	16
Mathematical Functions for complex	16
Trigonometric Functions for complex	17
Magnitude Functions for complex	18
Conversion Functions for complex	18
Using the c_exception Class to Handle Complex Mathematics Errors	19
Defining a Customized complex_error Function	19
Errors Handled Outside of the Complex Mathematics Library	20
An Example of Using the Complex Mathematics Library	20

Part 2. The I/O Stream Class Library 23

Chapter 3. Introduction to the I/O Stream Classes	25
The I/O Stream Classes and stdio.h	25
Overview of the I/O Stream Classes	25
Combining Input and Output of Different Types	26
Input and Output for User-Defined Classes	26
The I/O Stream Class Hierarchy	26
The I/O Stream Header Files	28
Predefined Streams	28
Anonymous Streams	29
Stream Buffers	30
What Does a Stream Buffer Do?	30
Why Use a Stream Buffer?	30
How Is a Stream Buffer Implemented?	30
Format State Flags	32
Thread Safety	32
Chapter 4. Getting Started with the I/O Stream Library	35
Receiving Input from Standard Input	35
Multiple Variables in an Input Statement	35
String Input	36
White Space in String Input	36
Incorrect Input and the Error State of the Input Stream	38
Using Input Streams Other Than cin	38
Displaying Output on Standard Output or Standard Error	38
Multiple Variables in an Output Statement	39
Using Output Streams Other Than cout, cerr, and clog	39
Flushing Output Streams with endl and flush	40
Placing endl or flush in an Output Stream	41
Parsing Multiple Inputs	41
Opening a File for Input and Reading from the File	42
Constructing an fstream or ifstream Object for Input	43
Reading Input from a File	44
Opening a File for Output and Writing to the File	45

Chapter 5. Advanced I/O Stream Topics	47
Associating a File with a Standard Input or Output Stream	47
Using filebuf Functions to Move Through a File	48
Defining an Input Operator for a Class Type	50
Using the cin Stream in a Class Input Operator	51
Displaying Prompts in Input Operator Code	52
Defining an Output Operator for a Class Type	52
Class Output Operators and the Format State	53
Correcting Input Stream Errors	54
Changing the Formatting of Stream Output	56
ios Methods and Manipulators	56
Using setf, unsetf, and flags	57
Changing the Notation of Floating-Point Values	59
Changing the Base of Integral Values	60
Setting the Width and Justification of Output Fields	61
Defining Your Own Format State Flags	61
Using the stringstream Classes for String Manipulation	63
 Chapter 6. Manipulators	65
Introduction to Manipulators	65
Simple Manipulators and Parameterized Manipulators	65
Creating Simple Manipulators for Your Own Types	66
Creating Parameterized Manipulators for Your Own Types	67

Part 3. The Collection Class Library 71

Chapter 7. Overview of the Collection Class Library	73
Benefits of the Collection Class Library	73
Concrete Classes Provided by the Library	73
Types of Classes in the Collection Class Library	77
Flat Collections	78
Ordering of Collection Elements	79
Access by Key	79
Equality for Keys and Elements	79
Uniqueness of Entries	81
Restricted Access	81
Trees	82
Auxiliary Classes	83
The Overall Implementation Structure	83
Categories of Classes	84
Default Classes	85
Variant Classes	85
Collection Class Hierarchy	85
Typed and Typeless Implementation Classes	85
Class Template Naming Conventions	86
 Chapter 8. Instantiating and Using the Collection Classes	89
Instantiation and Object Definition	89
Adding, Removing, and Replacing Elements	90
Adding Elements	90
Removing Elements	91
Replacing Elements	93
Cursors	93

Using Cursors for Locating and Accessing Elements	94
Iterating over Collections	96
Iteration Using Cursors	96
Iteration Using allElementsDo	97
Iteration Using Applicators	98
Copying and Referencing Collections	99
Bounded and Unbounded Collections	100
 Chapter 9. Element Functions and Key-Type Functions	101
Introduction to Element Functions and Key-Type Functions	101
Using Member Functions	102
Using Separate Functions	103
Using Element Operation Classes	105
Memory Management with Element Operation Classes	109
Functions for Derived Element Classes	109
Using Smart Pointers	111
Overview of Smart Pointers	112
Element Pointers	113
Managed Pointers	116
Automatic Pointers	116
Constructing Smart Pointers	118
 Chapter 10. Tailoring a Collection Implementation	121
Introduction	121
Replacing the Default Implementation	121
The Based-On Concept	122
Provided Implementation Variants	122
Features of Provided Implementation Variants	123
Sequences	124
Trees	126
Hash Table	129
 Chapter 11. Polymorphism and the Collections	131
Introduction to Polymorphism	131
Using the Abstract Class Hierarchy	131
Adding and Overloading Member Functions	132
 Chapter 12. Support for Notifications	135
Example for IVSequence<IString>	136
 Chapter 13. Thread Safety and the Collection Classes	139
Guard Objects	139
Usage	139
Restrictions	141
 Chapter 14. Exception Handling	143
Introduction to Exception Handling	143
Exceptions Caused by Violated Preconditions	143
Exceptions Caused by System Failures and Restrictions	144
Precondition and Defined Behavior	144
Levels of Exception Checking	145
List of Exceptions	145
The Hierarchy of Exceptions	147

Chapter 15. Collection Class Library Tutorials	149
Preparing for the Lessons	150
Lesson 1: Defining a Simple Collection of Integers	150
Lesson 2: Adding, Listing, and Removing Elements	153
Lesson 3: Changing the Element Type	158
Lesson 4: Changing the Collection	163
Lesson 5: Changing the Implementation Variant	171
Errors When Compiling or Running the Lessons	173
Other Tutorials	173
Using the Default Classes	173
Advanced Use	174
Source Files for the Tutorials	174
 Chapter 16. Solving Problems in the Collection Class Library	177
Cursor Usage	177
Element Functions and Key-Type Functions	178
Key Access Function - How to Return the Key (1)	179
Key Access Function - How to Return the Key (2)	180
Definition of Key-Type Functions	180
Exception Tracing	181
Declaration of Template Arguments and Element Functions (1)	181
Declaration of Template Arguments and Element Functions (2)	181
Declaration of Template Arguments and Element Functions (3)	182
Default Constructor	182
 Chapter 17. Compatibility Information	185
Compatible Items	185
Incompatible Items	186

Part 4. Application Support Class Library 189

Chapter 18. Application Support Class Library	191
Organization of Classes	191
IBase Class	194
IVBase Class	194
String and Buffer Classes	195
Thread Safety	195
MBCS and National Language Support	195
Turning on Internationalization Semantics	196
Setting the Locale	196
 Chapter 19. String Classes	199
Introduction to the String Classes	199
String Buffers	200
Multiple-Byte Character Set Support	200
Indexing of Strings	200
What You Can Do with Strings	200
Creating and Copying Strings	201
Doing String Input and Output	203
Concatenating Strings	204
Finding Words or Substrings within Strings	204
Replacing, Inserting, and Deleting Substrings	206
Determining String Lengths and Word Counts	207

Extending Strings	207
Converting between Strings and Numeric Data	207
Converting between Strings and Different Base Notations	208
Testing the Characteristics of Strings	209
Formatting Strings	211
Other IString Capabilities	212
IStringTest Class	212
 Chapter 20. Exception and Trace Classes	215
Introduction to the Exception Classes	215
Characteristics of the Exception Classes	215
Derivation of the Exception Classes	215
Situations in Which the Exception Classes Are Used	216
Catching Exceptions Thrown by Class Library Functions	217
An Example of the Subscript Operator Throwing an Exception	217
Throwing Your Own Exceptions Using the Exception Classes	218
Macros Used with the Exception Classes	219
Why Use the Macros?	220
Using the ITrace Class	222
Using the Trace Macros to Control Trace Output	222
Capturing Trace Output in a File	223
An Example of Using ITrace	223
 Chapter 21. Date and Time Classes	225
IDate Class	225
Creating an IDate Object	225
Changing an IDate Object	226
Information Functions for IDate Objects	226
Testing and Comparing IDate Objects	226
ITime Class	227
Creating an ITime Object	227
Changing an ITime Object	227
Information Functions for ITime Objects	227
Comparing ITime Objects	228
Writing an ITime Object to an Output Stream	228
ITimeStamp Class	229
Creating an ITimeStamp Object	229
Changing an ITimeStamp Object	229
Information Functions for ITimeStamp Objects	229
Comparing ITimeStamp Objects	230
 Chapter 22. Controlling Threads and Protecting Data	231
Accessing the Current Thread	232
Starting a Thread	232
Starting Nonmember Functions	232
Starting a Member Function	232
Protecting Data	234
 Chapter 23. The IBM Open Class Notification Framework	235
Notifiers and Observers	235
Notification Protocol	236
IBM C++ Notification Class Hierarchy	237
 Chapter 24. Using the Binary Coded Decimal Class	239

Header File and Constants for IBinaryCodedDecimal	239
Constants Defined in idecimal.hpp	239
Constructing IBinaryCodedDecimal Objects	240
IBinaryCodedDecimal Input and Output	240
Mathematical Operators for IBinaryCodedDecimal	240
Relational Operators	240
Equality Operators	241
Converting IBinaryCodedDecimal Objects	241
IBinaryCodedDecimal Object to a IBinaryCodedDecimal Object	241
Number of Digits of an IBinaryCodedDecimal Object	242
Precision of an IBinaryCodedDecimal Object	242
IBinaryCodedDecimal Object Exceptions	242
 Chapter 25. Using the Decimal Class	 243
Header File	243
Constructing Decimal Objects	243
Decimal Class Input and Output	244
Operators for Decimal Class	244
Arithmetic Operators	244
Relational Operators	245
Equality Operators	245
Converting Decimal Objects	245
Decimal Object to a Decimal Object	245
Decimal Object to an IString Object	246
Decimal Object from a char * Type	246
Decimal Object from an Integer Type	246
Decimal Object to and from IBinaryCodedDecimal Object	246
Number of Digits in a Decimal Object	246
Precision of a Decimal Object	246
Decimal Object Exceptions	247

Part 5. Glossary, Bibliography and Index 249

Glossary	251
 Bibliography	 281
OS/390	281
VS COBOL II Release 4	281
COBOL FOR MVS & VM Release 2	281
COBOL for OS/390 & VM Version 2 Release 1	281
PL/I for MVS & VM Release 1 Modification 1	281
OS PL/I Version 2 Release 3	282
VS FORTRAN Version 2 Release 6	282
CICS/ESA Version 4 Release 1	282
CICS Transaction Server for OS/390 Release 2	282
DB2 Version 3 Release 1	282
DB2 Version 4 Release 1	282
DB2 Version 5 Release 1	282
IMS/ESA Version 4 Release 1	282
IMS/ESA Version 5 Release 1	282
IMS/ESA Version 6 Release 1	283
QMF Version 3 Release 2	283
VSAM	283

Index 285

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 gives you guidance on how to use 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, beginning with an overview of IBM Open Class Library, and followed by a part for each of the class libraries listed above.

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
<p><i>OS/390 C/C++ Programming Guide</i>, SC09-2362</p>	<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++
<p><i>OS/390 C/C++ User's Guide</i>, SC09-2361</p>	<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.

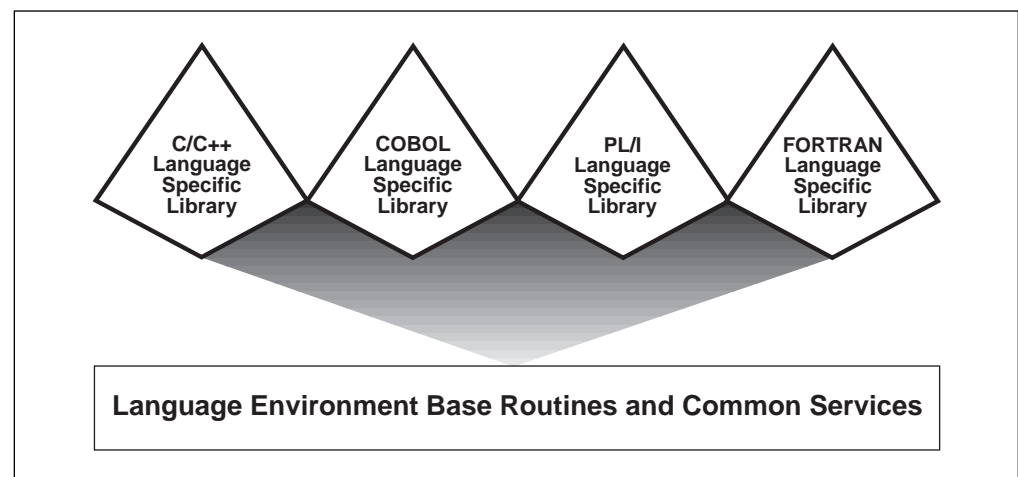


Figure 1. Libraries in OS/390 Language Environment

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>wcsrombs()</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.

Chapter 1. Introduction to IBM Open Class Library

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

- The *Complex Mathematics Class Library* provides you with the facilities to manipulate complex numbers and perform standard mathematical operations on them.
- The *I/O Stream Class Library* gives you the facilities to deal with many varieties of input and output. You can derive classes from I/O Stream classes to customize the input and output facilities for your own particular needs.
- The *Collection Class Library* provides a set of commonly used abstract data types that you can use to build collections. Collections can have properties such as sorted or unsorted, ordered or unordered, unique-element or multiple-element.
- The *Application Support Class Library* provides a set of classes that let you manipulate string, date, time, and timestamp information, create binary coded decimal objects, handle notifications, and trace exceptions. Thread and resource locking support is available in an OS/390 Unix System Services environment.

History of IBM Open Class Library

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

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

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

Hierarchies of the Class Libraries

The following figures show the class hierarchy of the class libraries that make up IBM Open Class. Some of these figures are repeated in the parts that describe specific libraries.

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

Class Library Hierarchies

The following Application Support classes are not shown because they do not derive from any class and do not have any subclasses:

- IExceptionLocation
- IMessageText
- IStringEnum
- IException::TraceFn
- IBinaryCodedDecimal
- IDecimalUtil

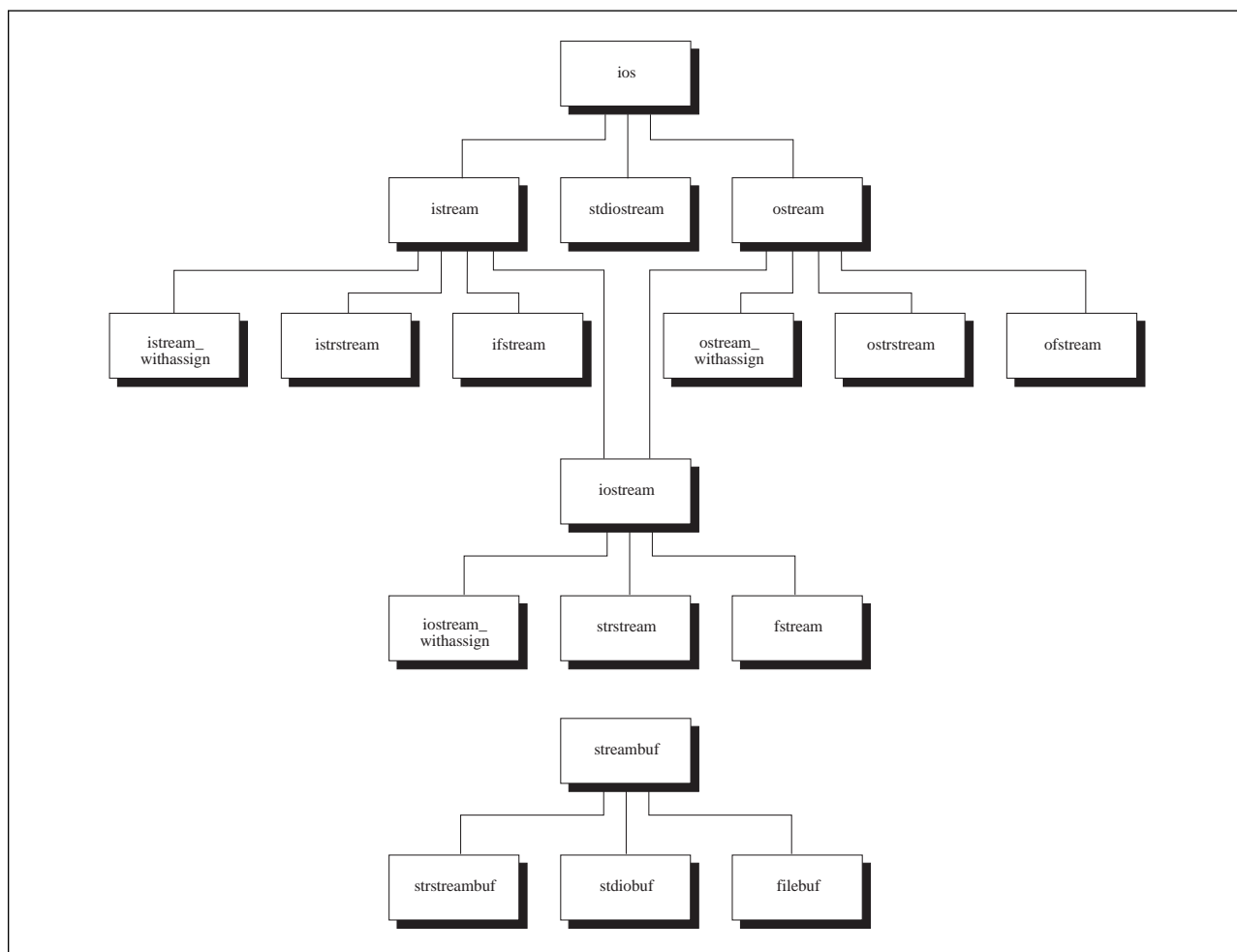


Figure 2. I/O Stream Library Hierarchy

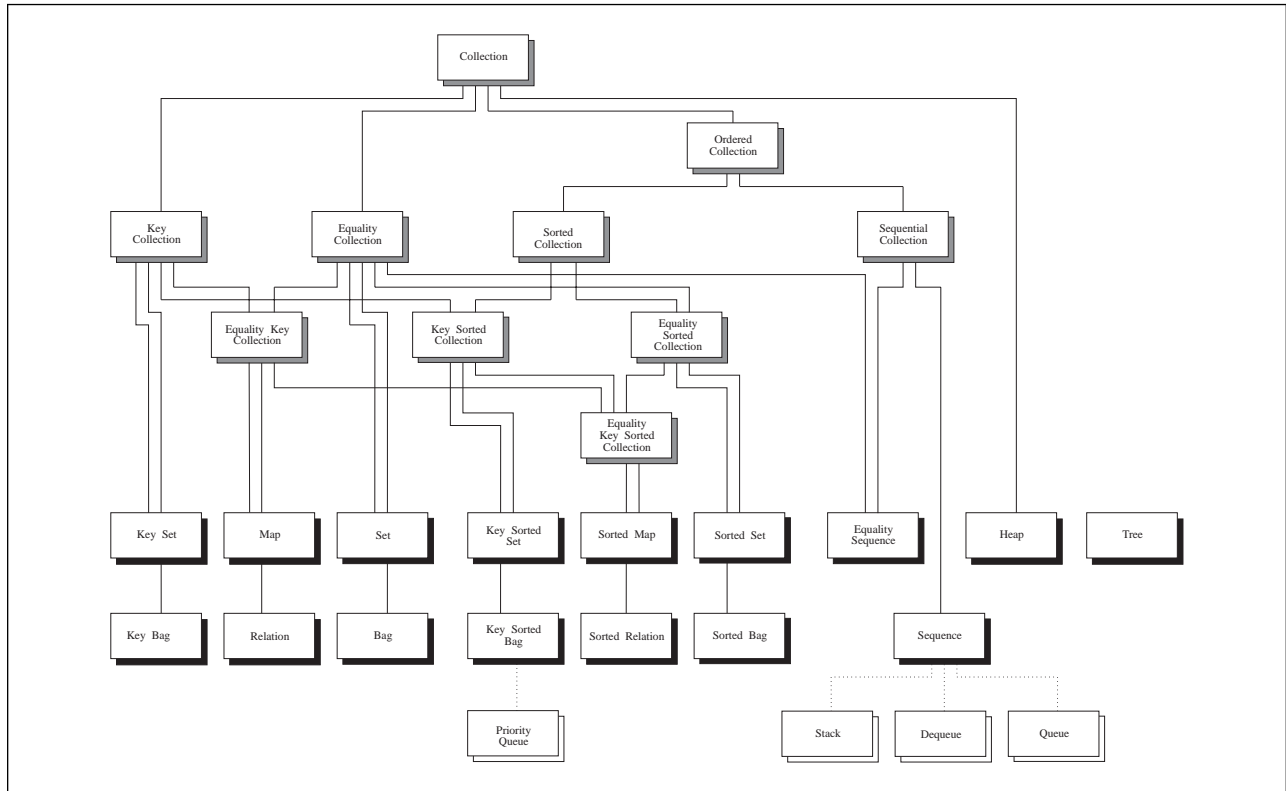


Figure 3. Collection Class Library Hierarchy. Abstract classes have a grey background. Concrete classes have a black background. Restricted access classes have a white background. Dotted lines show a “based-on” relationship, not an actual derivation.

Class Library Hierarchies

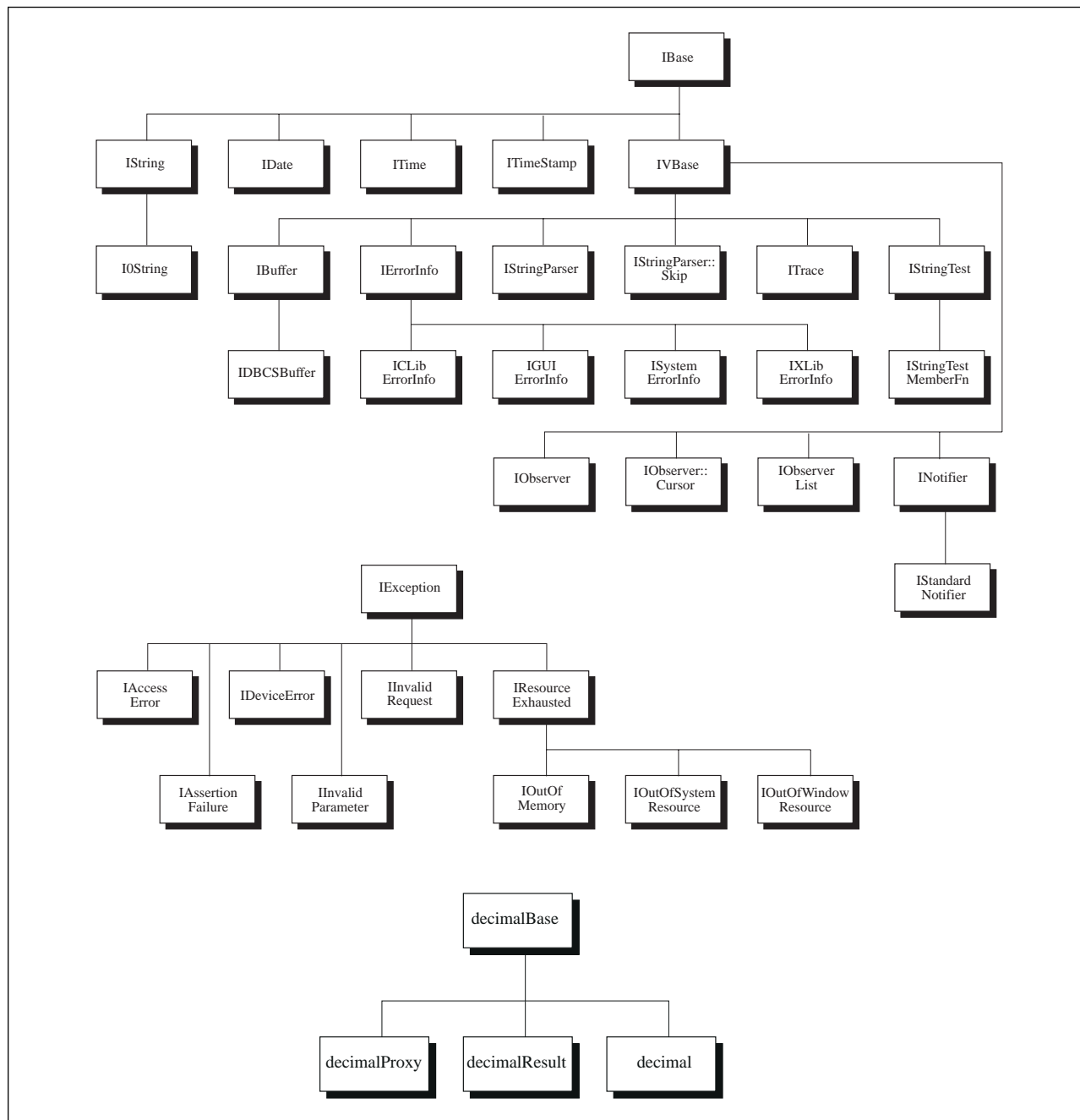


Figure 4. Application Support Class Hierarchy. Some class names have been split into two lines to fit in their boxes. Note that IGUIErrorInfo, IXLlibErrorInfo, and IOutOfSystemResource are not supported on OS/390 C/C++. The classes IDecimalUtil, decimalBase, decimalProxy, and decimalResult are meant for internal use of the Application Support Classes. Do not use them directly.

This release includes three types of class libraries:

- C++ native
- C++ SOM, which provides release-to-release binary compatibility (RRBC).
- Cross-language SOM, which provides RRBC and cross-language support.

The I/O Stream and Complex Class Libraries are available in C++ native versions only.

The Application Support Class Library is available in:

- C++ native
- C++ SOM

The Collection Class Library is available in:

- C++ native
- C++ SOM
- Cross-language SOM

All of these class libraries are available in both static and DLL forms.

The C++ native versions of the libraries are documented in this book and the *OS/390 C/C++ IBM Open Class Library Reference*.

You use the C++ SOM class libraries the same way as the C++ native versions, except that you compile and link them differently. Instructions for compiling and linking the C++ SOM class libraries can be found in the *OS/390 C/C++ SOM-Enabled Class Library User's Guide and Reference*; for guidance and reference information on these libraries, see this book and the *OS/390 C/C++ IBM Open Class Library Reference*.

All information for the cross-language SOM Collection Class Library is included in the *OS/390 C/C++ SOM-Enabled Class Library User's Guide and Reference*.

Coding with Class Libraries under OS/390 UNIX System Services

If you use the class libraries in applications that run under OS/390 UNIX System Services, the following restrictions apply:

- These class libraries are not safe to use with respect to asynchronous signals. In particular, this means:
 - Do not invoke these classes from a signal handler during asynchronous signal handling.
 - Do not call `longjmp()` or `siglongjmp()` from a signal handler during asynchronous signal handling. If you do, the class library behavior is undefined.

If you don't observe these restrictions, the subsequent behavior of any of these classes is undefined. For simplicity, avoid invoking these classes from any signal handler.

- Do not call `fork()` in user code that has been invoked from class library code (for example, in an override of `IStringTest::test` from the Application Support Class library or in an implementation of `allElementsDo` from the Collection Class library).

Compiling Programs that Use IBM Open Class Library

If your source code includes the IBM-supplied class library header files, you must use the `SEARCH` compiler option to identify the relevant data sets. Using the `SYSLIB DDname` may result in compilation errors.

The IBM-supplied cataloged procedures, REXX EXECs and panels include the standard header file data sets and the class library header file data sets on the default `SEARCH`.

Your search path should look like:

```
SEARCH ('CEE.SCEEH.+' , 'CBC.SCLBH.+' )
```

This will provide access to the following data sets:

- CEE.SCEEH.H (standard header files)
- CBC.SCLBH.H (class library header files)
- CBC.SCLBH.C (class library files)
- CBC.SCLBH.INL (class library files)
- CBC.SCLBH.HPP (application support class library headers)

Note: Do not use the class library source data sets (CBC.SCLDH.*) unless you are using your own libraries built from the source in CBC.ASCSRC or CBC.CCLSRC. If you are, then the class library source data sets must be specified first in the search order.

Binding with IBM Open Class Library

The IBM-supplied catalog procedures links the C++ DLL versions of IBM Open Class Library by default. The binder input definition side-decks are in data set CBC.SCLBSID, members COMPLEX, IOSTREAM, and ASCCOLL. If you want to statically bind the Open Class object code instead, you can override the BIND.SYSLIB concatenation to include the CBC.SCLBCPP data set, and override the BIND.SYSIN concatenation to exclude the CBC.SCLBSID members.

The DLLs are in data set CBC.SCLBDLL members COMPLEX (Complex Mathematics Class Library), IOSTREAM (I/O Stream Class Library), and ASCCOLL (Collection Class Library and Application Support Class Library). This data set must be available at run time as it contains the class library messaging modules, as well as the DLLs.

The CEE.SCEERUN data set must also be available at run time. These data sets can be in the system libraries, your JOBLIB statement or your STEPLIB statement.

Note: Your application cannot use multiple copies of an IBM Open Class library. If your application consists of multiple modules (for example, a main module and a DLL) that use the same class library, make sure that all your modules link dynamically to the class library. Otherwise, the class library will be linked in multiple times, and there will be multiple copies in use by your application. The use of multiple copies of a class library within a single application is not supported, and can have unexpected results.

See the *OS/390 C/C++ User's Guide* for more information on compiling and binding options. For information on compiling and binding with the C++ SOM Library, see the *OS/390 C/C++ SOM-Enabled Class Library User's Guide and Reference*.

Migration Notes

The Collection Class Library and the Application Support Class Library DLLs and side-decks have been merged in OS/390 C/C++ V1R3M0. The combined DLL and side-deck is ASCCOLL. This single side-deck and DLL should be used for all new applications.

Side-decks with the names APPSUPP (Application Support Class Library entry points only) and COLLECT (Collection Class Library entry points only) are supplied for migration purposes only. These side-decks should only be used in circumstances where a name collision exists between your application code and one of the

libraries. For example, the application uses the Application Support Class Library and contains a function with the same name as one in the Collection Class Library. In this example, you must prelink with the APPSUPP side-deck only to allow the duplicate name to be resolved in the application code. The combined DLL, ASCCOLL, will still be used.

DLLs with the names APPSUPP and COLLECT are provided for applications linked with previous releases of OS/390 C/C++. These DLLs are equivalent to ASCCOLL.

Thread Safety and the IBM Open Class Library

A *thread* is an independent, lightweight control activity within a computer process. In a multithreaded application, many threads typically exist within a single process and all share the same address space.

Thread safety for an application or function is the ability of that application or function to run in a multithreaded environment. An application or function that is not thread safe is not guaranteed to run correctly in a multithreaded environment, even if it does not itself directly employ multithreading. However, a thread safe application or function does not relieve programmers of the responsibility to properly manage their own resources.

Why Thread Safety?

Global instances of IBM Open Class Library classes often result in global data structures that are shared among all threads in a process. In a multithreaded environment, unrestricted access to these global data structures can result in unpredictable behavior. To avoid problems, you need to ensure that access to global data structures or resources is serialized so that no two threads can access the resources simultaneously. This is accomplished by protecting the resources under a lock.

Levels of Thread Safety

From an application viewpoint, there are three levels of thread safety:

Level 0 No safety. It is only safe to instantiate a given class on a single thread.

Level 1 Class safety. It is safe to instantiate a given class on multiple threads. This implies that constructors are safe, global class data is implicitly serialized, and instances are safe if they are not shared across threads. With Level 1 thread safety, programmers are required to define the critical regions within their programs.

While it is generally safe to instantiate a given class on different threads, sharing a specific object across threads requires explicit program serialization and coordination.

As a comparison, consider fundamental types such as `int` and `float`. For these types, it is safe to define different variables of a given type on multiple threads, but sharing a given variable across threads also requires explicit programmer serialization and coordination.

Level 2 Instance safety. It is safe to use a given instance on separate threads. This implies Level 1 thread safety with the added feature that instance data is implicitly serialized. Explicit programmer serialization or coordination is not required.

Thread Safety and the IBM Open Class Library

In general, the IBM Open Class Library is at Level 1 thread safety. The notable exceptions are the I/O Stream and Collection Class libraries. The I/O Stream Library is at Level 2 thread safety. Simultaneous use of `cout << ...` from multiple threads can be accomplished without programmer serialization. The Collection Class Library, while at Level 1 thread safety, provides an assist for explicit programmer serialization in the form of a Guard class. Information about Collection Class Library thread safety and the Guard class is found in Chapter 13, "Thread Safety and the Collection Classes" on page 139.

Part 1. Complex Mathematics Class Library

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

Chapter 2. Using the Complex Mathematics Classes	11
Review of Complex Numbers	11
Header Files and Constants for <code>complex</code> and <code>c_exception</code>	12
Constructing complex Objects	12
Complex Mathematics Input and Output	13
Mathematical Operators for <code>complex</code>	14
Friend Functions for <code>complex</code>	16
Using the <code>c_exception</code> Class to Handle Complex Mathematics Errors	19
Errors Handled Outside of the Complex Mathematics Library	20
An Example of Using the Complex Mathematics Library	20

Chapter 2. Using the Complex Mathematics Classes

This chapter reviews the concept of complex numbers, and then describes `complex`, the class you use to manipulate complex numbers, and `c_exception`, the class you use to handle errors 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.

Review of Complex Numbers

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

- $(a,b) + (c,d) = (a+c,b+d)$
- $(a,b) - (c,d) = (a-c,b-d)$
- $(a,b) * (c,d) = (ac-bd,ad+bc)$
- $(a,b) / (c,d) = ((ac+bd) / (c^2+d^2), (bc-ad) / (c^2+d^2))$
- The conjugate of a complex number (a,b) is $(a,-b)$
- The absolute value or *magnitude* of a complex number (a,b) is the positive square root of the value $a^2 + b^2$
- The polar representation of (a,b) is $(r,theta)$, where r is the distance from the origin to the point (a,b) in the complex plane, and $theta$ is the angle from the real axis to the vector (a,b) in the complex plane. The angle $theta$ can be positive or negative. Figure 5 illustrates the polar representation $(r,theta)$ of the complex number (a,b) .

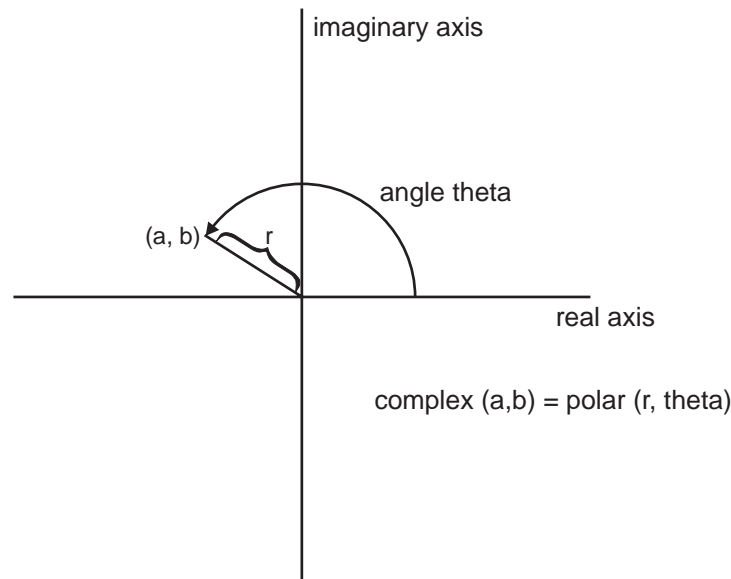


Figure 5. Polar Representation of Complex Number (a,b)

Header Files and Constants for complex and c_exception

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

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

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

Constants Defined in `complex.h`

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

Table 3. 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
<code>M_SQRT2</code>	The square root of 2
<code>M_SQRT1_2</code>	The square root of $1 / 2$

Constructing complex Objects

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

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

Complex Mathematics Input and Output

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

CLB3ACOM

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

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

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

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

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

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

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

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

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

Mathematical Operators for complex

```
Here is scalar d: ( 17, 0)
Here is the address of array a:
0x2ff7f9b8
```

Mathematical Operators for complex

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

CLB3AMTO

```
// Using the complex mathematical operators

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

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

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

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

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

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

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

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

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

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

```
Enter six complex numbers, separated by spaces:
(1.14,2.28) (2.24,4.48) (1.17,12.18)
(4.444444,5.12341) (12,7) 5
Here are the seven numbers after calculations:
a=( -10.86, -4.72)
b=( 1.14, 2.28)
c=( 2.24, 4.48)
d=( 6.17, 12.18)
e=( 4.44444, 5.12341)
f=( 12, 7)
g=( 5, 0)
```


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

Equality and Inequality Operators Test for Absolute Equality

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

CLB3AEQU

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

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

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

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

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

Friend Functions for complex

This example produces the following output:

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

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

Assignment Operators Do Not Produce an lvalue

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

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

Friend Functions for complex

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

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

Mathematical Functions for complex

The complex class defines four mathematical functions as friend functions of complex objects. The functions, described in detail in the *OS/390 C/C++ IBM Open Class Library Reference*, are:

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

The following example shows uses of these mathematical functions:

CLB3AMTH

```
// Using the complex mathematical functions

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

void main() {
    complex a, b;
    int i;
    double f;
    //
    // prompt the user for an argument for calls to
    // exp(), log(), and sqrt()
    //
    cout << "Enter a complex value\n";
    cin >> a;
```

```

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

```

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

```

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

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

```

Trigonometric Functions for complex

The complex class defines four trigonometric functions as friend functions of complex objects. The functions, described in detail in the *OS/390 C/C++ IBM Open Class Library Reference*, are:

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

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

CLB3ATRG

```

// Complex Mathematics Library trigonometric functions

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

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

```

This program produces the following output:

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

Magnitude Functions for complex

The magnitude functions for complex are:

- abs - Absolute value
- norm - Square magnitude

See the *OS/390 C/C++ IBM Open Class Library Reference* for further details on these functions.

Conversion Functions for complex

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

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

- arg - Angle in radians
- conj - Conjugation
- polar - Polar to complex
- real - Extract real part
- imag - Extract imaginary part

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

CLB3ACNV

```
// Using the complex conversion functions

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

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

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

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

Using the `c_exception` Class to Handle Complex Mathematics Errors

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

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

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

If you do not define your own `complex_error` function, `complex_error` sets the complex return value and the `errno` error number as defined in Table 4 in the *OS/390 C/C++ IBM Open Class Library Reference*.

Defining a Customized `complex_error` Function

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

CLB3ACER

```
//Redefinition of the complex_error function

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

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

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

This example produces the following output:

```
=====
Exception
type = 3
name = exp
arg1 = ( 7.23701e+75, 0)
arg2 = ( 0, 0)
retval = ( 7.23701e+75, -7.23701e+75)
=====
exp( 7.23701e+75, 0)= ( 7.23701e+75, 7.23701e+75)
```

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

```
exp( 7.23701e+75, 0) = ( 7.23701e+75, -7.23701e+75)
```

Compiling a Program that Uses a Customized `complex_error` Function

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.

Errors Handled Outside of the Complex Mathematics Library

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

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

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

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

An Example of Using the Complex Mathematics Library

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

$$\begin{aligned} &\sigma \\ &\sigma \cdot \omega \\ &\sigma \cdot \omega^2 \\ &\sigma \cdot \omega^3 \\ &\cdot \\ &\cdot \\ &\cdot \\ &\sigma \cdot \omega^{n-1} \end{aligned}$$

where ω is a complex number with the value:

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

and σ is a complex number with the value:

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

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

CLB3AROT

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

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

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

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

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

void main() {
    double n;
    complex input, omega, sigma;
    //
    // prompt the user for a complex number
    //
    cout << "Please enter a complex number: ";
    cin >> input;
    //
    // prompt the user for the value of n
    //
    cout << "What root would you like of this number? ";
    cin >> n;
```

Complex Mathematics Library Example

```
//  
// calculate the value of omega  
//  
omega = get_omega(n);  
cout << "Here is omega " << omega << endl;  
//  
// calculate the value of sigma  
//  
sigma = get_sigma(input,n);  
cout << "Here is sigma " << sigma << '\n'  
    << "Here are the " << n << " roots of " << input << endl;  
for (int i = 0; i < n ; i++) {  
    cout << sigma*(pow(omega,i)) << endl;  
}  
}
```

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

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

Part 2. The I/O Stream Class Library

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

Chapter 3. Introduction to the I/O Stream Classes	25
The I/O Stream Classes and <code>stdio.h</code>	25
Overview of the I/O Stream Classes	25
The I/O Stream Class Hierarchy	26
The I/O Stream Header Files	28
Predefined Streams	28
Anonymous Streams	29
Stream Buffers	30
Format State Flags	32
Thread Safety	32
 Chapter 4. Getting Started with the I/O Stream Library	 35
Receiving Input from Standard Input	35
Displaying Output on Standard Output or Standard Error	38
Flushing Output Streams with <code>endl</code> and <code>flush</code>	40
Parsing Multiple Inputs	41
Opening a File for Input and Reading from the File	42
Opening a File for Output and Writing to the File	45
 Chapter 5. Advanced I/O Stream Topics	 47
Associating a File with a Standard Input or Output Stream	47
Using <code>filebuf</code> Functions to Move Through a File	48
Defining an Input Operator for a Class Type	50
Defining an Output Operator for a Class Type	52
Correcting Input Stream Errors	54
Changing the Formatting of Stream Output	56
Defining Your Own Format State Flags	61
Using the <code>stringstream</code> Classes for String Manipulation	63
 Chapter 6. Manipulators	 65
Introduction to Manipulators	65
Simple Manipulators and Parameterized Manipulators	65
Creating Simple Manipulators for Your Own Types	66
Creating Parameterized Manipulators for Your Own Types	67

Chapter 3. Introduction to the I/O Stream Classes

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

The I/O Stream Classes and `stdio.h`

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

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

Overview of the I/O Stream Classes

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

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

1. The `streambuf` class and the classes derived from it (`strstreambuf`, `stdiobuf`, and `filebuf`) implement the *stream buffers*. Stream buffers act as temporary repositories for characters that are coming from the *ultimate producers* of input or are being sent to the *ultimate consumers* of output. See “Stream Buffers” on page 30 for more details.
2. The `ios` class maintains formatting and error-state information for these streams. The classes derived from `ios` implement the formatting of these streams. This formatting involves converting sequences of characters from the stream buffer into values of a particular type and converting values of a particular type into their external display format.

The I/O Stream Classes predefine streams for standard input, standard output, and standard error. See “Predefined Streams” on page 28 for more details on the predefined streams. If you want to open your own streams for input or output, you must create an object of an appropriate I/O Stream class. The `iostream` constructor takes as an argument a pointer to a `streambuf` object. This object is

associated with the device, file, or array of bytes in memory that is going to be the ultimate producer of input or the ultimate consumer of output.

Combining Input and Output of Different Types

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

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

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

Input and Output for User-Defined Classes

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

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

The I/O Stream Class Hierarchy

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

- The `streambuf` class implements *stream buffers*. See “Stream Buffers” on page 30 for information on how and why to use stream buffers. `streambuf` is the base class for the following classes:
 - `strstreambuf`
 - `stdiobuf`
 - `filebuf`
- The `ios` class maintains formatting and error state information for streams. Streams are implemented as objects of the following classes that are derived from `ios`:
 - `stdiostream`
 - `istream`
 - `ostream`

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

- `istream` is the input stream class. It implements stream buffer input, or *input* operations. The following classes are derived from `istream`:
 - `istrstream`
 - `ifstream`
 - `istream_withassign`
 - `iostream`

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

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

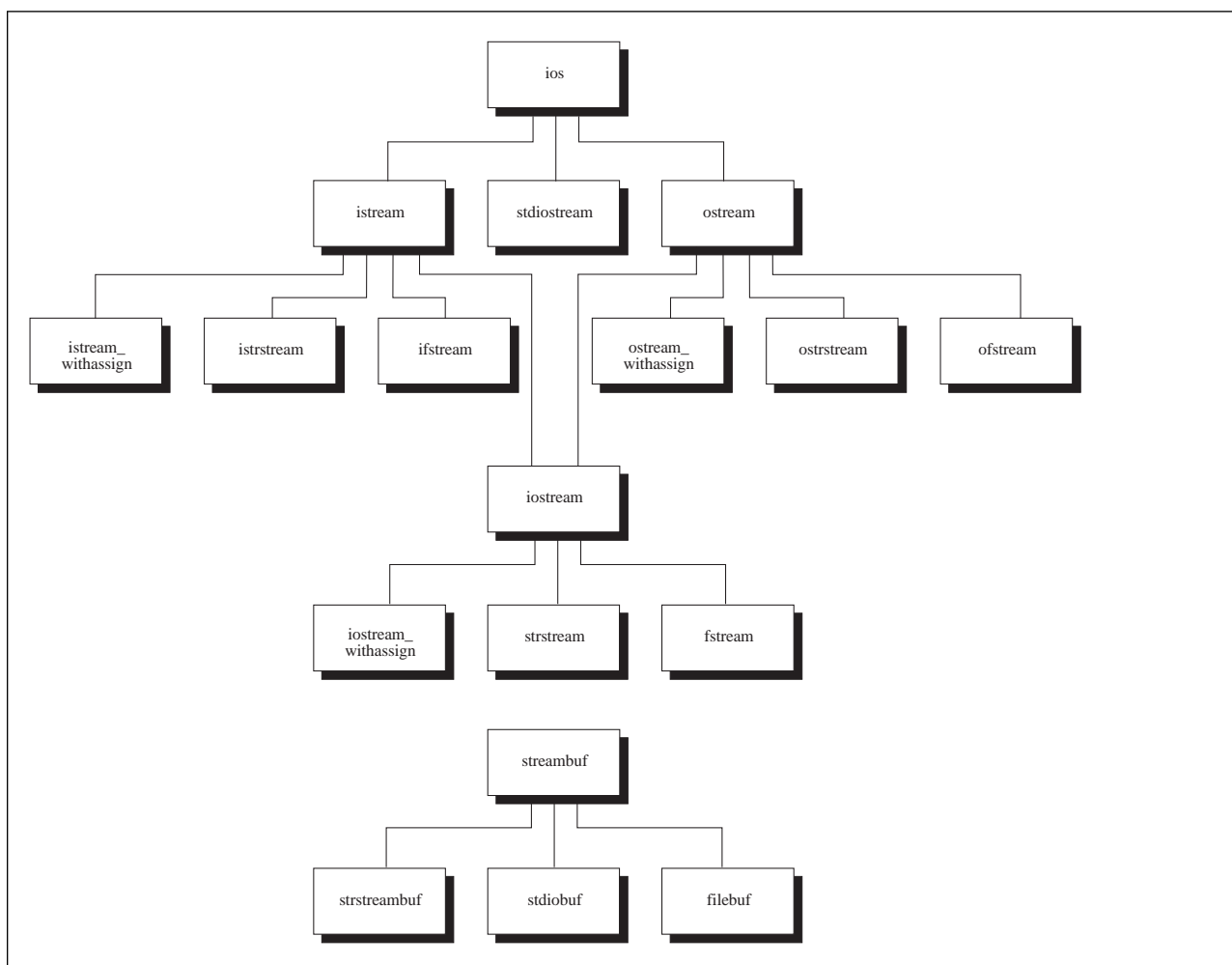


Figure 6. I/O Stream Class Hierarchy

Figure 6 shows the relationship between the two base classes, `ios` and `streambuf`, and their derived classes. In the figure, for any two classes connected by a line, the class at the lower level is derived from the class at the higher level.

The I/O Stream Header Files

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

- `iostream.h` contains declarations for the basic classes:
 - `streambuf`
 - `ios`
 - `istream`
 - `istream_withassign`
 - `ostream`
 - `ostream_withassign`
 - `iostream`
 - `iostream_withassign`
- `fstream.h` contains declarations for the classes that deal with files:
 - `filebuf`
 - `ifstream`
 - `ofstream`
 - `fstream`
- `stdiostream.h` contains declarations for `stdiobuf` and `stdiostream`, the classes that specialize `streambuf` and `ios`, respectively, to use the `FILE` structures defined in the C header file `stdio.h`.
- `strstream.h` contains declarations for the classes that deal with character strings. The first “str” in each of these names stands for “string”:
 - `istrstream`
 - `ostrstream`
 - `strstream`
 - `strstreambuf`
- `iomanip.h` contains declarations for the parameterized manipulators. Manipulators are values that you can insert into streams or extract from streams to affect or query the behavior of the streams.
- `stream.h` is used for compatibility with earlier versions of the I/O Stream Classes. It includes `iostream.h`, `fstream.h`, `stdiostream.h`, and `iomanip.h`, along with some definitions needed for compatibility with the ATT C++ Language System Release 1.2. Only use this header file with existing code; do not use it with new C++ code.

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

Predefined Streams

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

- `cin` is the standard input stream.
- `cout` is the standard output stream.

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

The predefined streams `cin`, `cerr`, and `clog` are *tied* to `cout`. As a result, if you use `cin`, `cerr`, or `clog`, `cout` is *flushed*. That is, the contents of `cout` are sent to their ultimate consumer. See “tie” in the *OS/390 C/C++ IBM Open Class Library Reference* for more details on tying streams together.

Anonymous Streams

An *anonymous stream* is a stream that is created as a temporary object. Because it is a temporary object, an anonymous stream requires a **const** type modifier and is not a modifiable lvalue. Unlike the ATT C++ Language System Release 2.1, the OS/390 C++ compiler does not allow a non-**const** reference argument to be matched with a temporary object. User-defined input and output operators usually accept a non-**const** reference (such as a reference to an `istream` or `ostream` object) as an argument. Such an argument cannot be initialized by an anonymous stream, and thus an attempt to use an anonymous stream as an argument to a user-defined input or output operator will usually result in a compile-time error.

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

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

CLB3ANON

```
// Using anonymous streams

#include <fstream.h>

class MyClass { public: char a; };

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

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

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

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

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

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

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

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

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

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

Stream Buffers

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

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

What Does a Stream Buffer Do?

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

Why Use a Stream Buffer?

The main reason for using stream buffers on OS/390 C/C++ is to ensure optimal portability.

How Is a Stream Buffer Implemented?

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

A stream buffer does not necessarily have separate get and put areas. A stream buffer that is used for input, such as one that is attached to an `istream` object, has

a get area. A stream buffer that is used for output, such as one that is attached to an `ostream` object, has a put area. A stream buffer that is used for both input and output, such as one that is attached to an `iostream` object, has both a get area and a put area. In stream buffers implemented by the `filebuf` class that are specialized to use files as an ultimate producer or ultimate consumer, the get and put areas overlap.

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

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

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

Figure 7 on page 32 shows how the pointers returned by these functions delineate the stream buffer.

Thread Safety

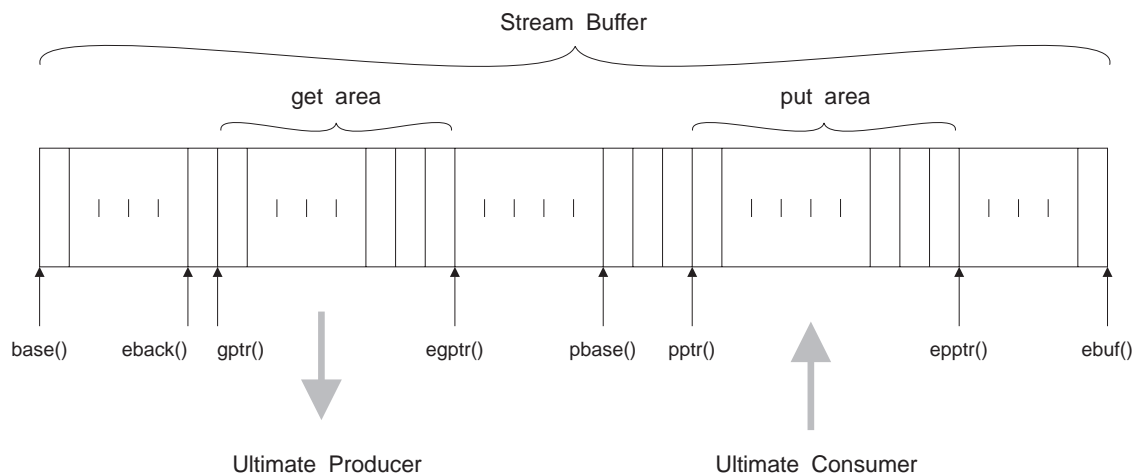


Figure 7. The Structure of Stream Buffers

Format State Flags

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

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

For examples of how to use these format state flags, see “Changing the Formatting of Stream Output” on page 56. For descriptions of individual format state flags, see “Format State Flags” in the *OS/390 C/C++ IBM Open Class Library Reference*.

Thread Safety

The I/O Stream Class Library provides thread safety at the object level. This means that it is safe to have multiple threads manipulate the same object.

The I/O Stream Class Library provides streaming operators for the built in C++ types. With object level thread safety, the output from one streaming operator will be streamed in entirety before the next. However, in a multi-threaded environment, there is no guarantee that the output from one streaming operator on the same thread will appear immediately after the output from the preceding streaming operator.

For example, given the following scenario, either result may occur:

Scenario:

```
thread 1: cout << anInt1 << aString1;  
thread 2: cout << anInt2 << aString2;
```

Result:

```
Desired:  anInt1 aString1 anInt2 aString2  
Possible: anInt1 anInt2 aString2 aString1
```

If order of output from separate threads is important, then explicit programmer serialization is required. For more information, see Chapter 22, “Controlling Threads and Protecting Data” on page 231.

Note: To run in a multi-threaded environment, the OS/390 UNIX kernel must be available and active.

Chapter 4. Getting Started with the I/O Stream Library

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

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

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

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

Receiving Input from Standard Input

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

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

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

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

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

Multiple Variables in an Input Statement

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

Receiving Input

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

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

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

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

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

String Input

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

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

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

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

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

White Space in String Input

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

CLB3AGET

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

#include <iostream.h>

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

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

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

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

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

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

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

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

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

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

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

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

Incorrect Input and the Error State of the Input Stream

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

Using Input Streams Other Than cin

You can use the same techniques for input from other input streams as for input from `cin`. The only difference is that, for other input streams, your program must define the stream. For information on how to define an input stream attached to a file, see "Opening a File for Input and Reading from the File" on page 42. Assuming you have defined a stream attached to a file opened for input, and have called that stream `myin`, you can read into that stream from the file by specifying that stream's name instead of `cin`:

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

Displaying Output on Standard Output or Standard Error

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

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

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

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

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

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

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

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

Multiple Variables in an Output Statement

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

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

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

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

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

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

Using Output Streams Other Than cout, cerr, and clog

You can use the same techniques for output to other output streams as for output to the predefined output streams. The only difference is that, for other output streams, your program must define the stream. For information on how to define an output stream attached to a file, see “Opening a File for Output and Writing to the File” on page 45. Assuming you have defined a stream attached to a file opened for output, and have called that stream `myout`, you can write to that file through its stream, by specifying the stream's name instead of `cout`, `cerr` or `clog`:

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

“Opening a File for Output and Writing to the File” on page 45 provides information on all operations required to perform basic file output, including opening, writing to, and closing output files.

Flushing Output Streams with endl and flush

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

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

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

A stream is flushed implicitly in the following situations:

- The predefined streams `cout` and `clog` are flushed when input is requested from the predefined input stream (`cin`).
- The predefined stream `cerr` is flushed after each output operation.
- An output stream that is unit-buffered is flushed after each output operation. A unit-buffered stream is a stream that has `ios::unitbuf` set. See “Buffer Flushing” in the *OS/390 C/C++ IBM Open Class Library Reference* for further details.
- An output stream is flushed whenever the `flush()` member function is applied to it. This includes cases where the `flush` or `endl` manipulators are written to the output stream. See “Placing endl or flush in an Output Stream” on page 41.
- The program terminates.

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

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

Placing endl or flush in an Output Stream

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

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

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

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

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

is not as efficient as:

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

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

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

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

Parsing Multiple Inputs

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

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

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

Input	Output	Remarks
123<\n>		No output. <code>a</code> has been assigned the value 123, but the program is still waiting on input for <code>b</code> and <code>c</code> .
1<\n> ,br 2<\n> 3<\n>	a: <1> b: <2> c: <3>	White space (in this case, new-line characters) is used to delimit different input variables.
1 2 3<\n>	a: <1> b: <2> c: <3>	White space (in this case, spaces) is used to delimit different input variables. There can be any amount of white space between inputs.
123,456,789<\n>	a: <123> b: <-559038737> c: <-559038737>	Characters are read into <code>int a</code> up to the first character that is not acceptable input for an integer (the comma). Characters are read into <code>int b</code> where input for <code>a</code> left off (the comma). Because a comma is not one of the allowable characters for integer input, <code>ios::failbit</code> is set, and all further input fails until <code>ios::failbit</code> is cleared. See "Correcting Input Stream Errors" on page 54 for details on how to clear an input stream.
1.2 2.3<\n> 3.4<\n>	a: <1> b: <-559038737> c: <-559038737>	As with the previous example, characters are read into <code>a</code> until the first character is encountered that is not acceptable input for an integer (in this case, the period). The next input of an <code>int</code> causes <code>ios::failbit</code> to be set, and so both it and the third input result in errors.

See "White Space in String Input" on page 36 for information on how the input operator interprets white space in the input stream during string input.

Opening a File for Input and Reading from the File

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

1. Construct an `fstream` or `ifstream` object to be associated with the file. The file can be opened during construction of the object, or later.

Note: OS/390 C/C++ provides overloads of the `fstream` and `ifstream` constructors and their `open()` functions, which allow you to specify file attributes such as `lrecl` and `recfm`. See the sections on these constructors and functions in the *OS/390 C/C++ IBM Open Class Library Reference* for further information.

2. Use the name of the `fstream` or `ifstream` object and the input operator or other input functions of the `istream` class, to read the input.
3. Close the file by calling the `close()` member function or by implicitly or explicitly destroying the `fstream` or `ifstream` object.

Constructing an `fstream` or `ifstream` Object for Input

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

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

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

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

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

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

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

CLB3AFST

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

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

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

```
barbers often shave
```

the file contains the following after the program is run:

```
barbers often shave
barbers
```

Note that you can use the same `fstream` object to access different files in sequence. In the above example, `myfile.open("test.C",ios::app)` could have

`read myfile.open("test.out",ios::app)` and the program would still have compiled and run, although the end result would be that the first string of `test.C` would be appended to `test.out` instead of to `test.C` itself.

Reading Input from a File

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

stdin.C	filein.C
<pre>#include <iostream.h> void main() { int ia,ib,ic; char ca[40],cb[40],cc[40]; // cin is predefined cin >> ia >> ib >> ic >> ca; cin.getline(cb,sizeof(cb),'\n'); cin >> cc; // no need to close cin cout << ia << ca << ib << cb << ic << cc << endl; }</pre>	<pre>#include <fstream.h> void main() { int ia,ib,ic; char ca[40],cb[40],cc[40]; fstream myfile("file.in",ios::in); myfile >> ia >> ib >> ic >> ca; myfile.getline(cb,sizeof(cb),'\n'); myfile >> cc; myfile.close(); cout << ia << ca << ib << cb << ic << cc << endl; }</pre>

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

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

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

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

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

Opening a File for Output and Writing to the File

The description of using a file as the input stream in “Opening a File for Input and Reading from the File” on page 42 provides the basis for explanations in this section. You may want to read that section first if you have not already done so.

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

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

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

You must specify one or more open modes when you open the file, unless you declare the object as an `ofstream` object. Open modes are described in “open” in the *OS/390 C/C++ IBM Open Class Library Reference*. The advantage of accessing an output file as an `ofstream` object rather than as an `fstream` object is that the compiler can flag input operations to that object as errors.

Note: OS/390 C/C++ provides overloads of the `fstream` and `ifstream` constructors and their `open()` functions, which allow you to specify file attributes such as `lrecl` and `recfm`. See the sections on these constructors and functions in the *OS/390 C/C++ IBM Open Class Library Reference* for further information.

2. Use the output operator or `ostream` member functions to perform output to the file.
3. Close the file using the `close()` member function of `fstream`.

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

Chapter 5. Advanced I/O Stream Topics

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

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

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

Associating a File with a Standard Input or Output Stream

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

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

CLB3AFLT

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

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

        if (argc>2) { // if an output file was also specified
            outfile = new ofstream(argv[2]); // try opening it
            if (outfile->good()) // if it opens successfully
                cout = *outfile; // assign output file to cout
        }
    }

    cin.getline(inputline,
        sizeof(inputline), '\n'); // get first line
```

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

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

Using filebuf Functions to Move Through a File

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

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

CLB3AFIL

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

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

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

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

    // declare an fstream object and open its file for input
    fstream InputFile("algonquin",ios::in);

    // check that input was successful, exit if not
    if (!InputFile) {
        cerr << "Could not open algonquin! Exiting...\n";
        exit(-1);
    }

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

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

```

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

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

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

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

```

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

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

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

the output of the example program is:

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

The following example shows how you can use both encoded and relative byte offsets to move through a file. Note that encoded offsets are specific to OS/390 C/C++ and programs that use them may not be portable to other VisualAge C++ (formerly C Set ++) compilers.

CLB3ATSE

```

// Example of using encoded and relative byte offsets
// in seeking through a file
#include <iomanip.h>
#include <fstream.h>

main() {
    fstream fs("tseek.data", ios::out); // create tseek.data
    filebuf* fb = fs.rdbuf();
    streamoff off[5];
    int pos[5] = {0, 30, 42, 197, 0};
}

```

Defining Your Own Input Operator

```
for (int i = 0, j = 0; i < 200; ++i) {
    if (i == pos[j])
        off[j++] = (*fb).seekoff(0L, ios::cur, ios::out);
    fs << setw(4) << i;
    if (i % 13 == 0 || i % 17 == 0) fs << endl;
}
fs.close();

cout << "Open the file in text mode, reposition using encoded\n"
     << "offsets obtained from previous calls to seekoff()" << endl;

fs.open("tseek.data", ios::in);
fb = fs.rdbuf();

// Exchange off[2] and off[3] so last seek will be backwards
off[4] = off[2]; off[2] = off[3]; off[3] = off[4];
pos[4] = pos[2]; pos[2] = pos[3]; pos[3] = pos[4];

for (j = 0; j < 4; ++j) {
    (*fb).seekoff(off[j], ios::beg, ios::in);
    fs >> i;
    cout << "data at pos" << dec << setfill(' ') << setw(4) << pos[j]
         << " is \"" << setw(4) << i << "\" (encoded offset was 0x"
         << hex << setfill('0') << setw(8) << off[j] << ")" << endl;
    if (i != pos[j]) return 37 + 10*j;
}
fs.close();
cout.fill(' ');
cout.setf(ios::dec, ios::basefield);

cout << "\nOpen the file in binary bytesseek mode, reposition using\n"
     << "byte offsets calculated by the user program" << endl;

fs.open("tseek.data", "bytesseek", ios::in|ios::binary);
fb = fs.rdbuf();

for (j = 0; j < 4; ++j) {
    off[j] = (*fb).seekoff(4*pos[j], ios::beg, ios::in);
    fs >> i;
    cout << "data at pos" << setw(4) << pos[j] << " is \"" << setw(4) << i
         << "\" (byte offset was " << setw(10) << off[j] << ")" << endl;
    if (i != pos[j]) return 77 + 10*j;
}
}
```

Defining an Input Operator for a Class Type

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

myclass.h

```
#include <iostream.h>

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

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

The input operator must have the following characteristics:

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

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

Using the `cin` Stream in a Class Input Operator

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

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

is rewritten as:

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

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

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

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

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

Displaying Prompts in Input Operator Code

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

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

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

Defining an Output Operator for a Class Type

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

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

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

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

The output operator must have the following characteristics:

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

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

Class Output Operators and the Format State

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

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

CLB3AOUT

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

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

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

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

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

Correcting Input Stream Errors

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

```
38.16221773.5917293.1
38.16221773.5917293.1
```

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

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

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

Correcting Input Stream Errors

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

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

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

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

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

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

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

All of these methods are described in Chapter 5, “ios Class” on page 31 *OS/390 C/C++ IBM Open Class Library Reference*.

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

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

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

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

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

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

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

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

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

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

Changing the Formatting of Stream Output

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

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

ios Methods and Manipulators

For some of the format flags defined for the `ios` class, you can set or clear them using an `ios` function and a flag name, or by using a manipulator. (Manipulators are described in more detail in Chapter 6, “Manipulators” on page 65). With manipulators you can place the change to a stream's state within a list of outputs for that stream. The following example shows two ways of changing the base of an output stream from decimal to octal. The first, which is more difficult to read, uses the `setf()` function to set the `basefield` field in the format state to octal. The second way uses a manipulator, `oct`, within the output statement, to accomplish the same thing.

```
#include <iostream.h>
void main() {
    int a=9;
    cout.setf(ios::oct,ios::basefield);
    cout << a << endl;
    // assume format state gets changed here, so we must change it back
    cout << oct << a << endl;
}
```

Note that you do not need to qualify a manipulator, provided you do not create a variable or function of the same name as the manipulator. If a variable `oct` were

declared at the start of the above example, `cout << oct ...` would write the variable `oct` to standard output. `cout << ios::oct ...` would change the format state.

Using `setf`, `unsetf`, and `flags`

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

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

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

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

1. The original settings of the format state flags are determined using `flags()`. These settings are saved in the variable `originalFlags`.
2. `ios::fixed` is set, and all other flags are cleared, using `flags(ios::fixed)`.
3. `ios::adjustfield` is set to `ios::right`, without affecting other fields, using `setf(ios::right)`.
4. `ios::floatfield` is set to `ios::scientific`, and `ios::adjustfield` is set to `ios::left`, without affecting other fields. The call to `setf()` is `setf(ios::scientific | ios::left, ios::floatfield|ios::adjustfield)`.
5. The original format state is restored by calling `flags()` with an argument of `originalFlags`, which contains the format state determined in step 1.

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

CLB3ASTF

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

#include <iostream.h>

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

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

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

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

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

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

This example produces the following output:

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

Note:

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

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

Changing the Notation of Floating-Point Values

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

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

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

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

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

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

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

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

CLB3AFLO

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

#include <iostream.h>

void main() {
    double a=3.14159265358979323846;
    double b;
    long originalFlags=cout.flags();
    int originalPrecision=cout.precision();
    for (double exp=1.;exp<1.0E+25;exp*=1000000000.) {
        cout << "Printing new value for b:\n";
        b=a*exp;    // Initialize b to a larger magnitude of a

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

The output from this program is:

```
Printing new value for b:
3.141593
3.141593e+00
3.141593E+00
3.141592653590E+00
Printing new value for b:
314159265.358979
3.141593e+08
3.141593E+08
3.141592653590E+08
Printing new value for b:
31415926535897932.000000
3.141593e+16
3.141593E+16
3.141592653590E+16
Printing new value for b:
31415926535897928000000000.000000
3.141593e+24
3.141593E+24
3.141592653590E+24
```

Changing the Base of Integral Values

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

CLB3ABAS

```
//Showing the base of integer values

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

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

Setting the Width and Justification of Output Fields

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

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

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

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

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

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

Defining Your Own Format State Flags

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

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

The program checks the input file for an exclamation mark as the first byte. If one is found, the input file is in the new format, and the output file should be in the old format. Otherwise the reverse is true. Once the program knows which file should be in which format, it requests a free flag from each file's stream object. It reads in

Defining Your Own Format State Flags

and writes out each record, and closes the file. The input and output operators for the class check the format state for the defined flag, and order their output accordingly.

CLB3ABIT

```
// Defining your own format flags

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

long InFileFormat=0;
long OutFileFormat=0;

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

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

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

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

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



```

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

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

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

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

```

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

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

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

Using the stringstream Classes for String Manipulation

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

Note: For new applications, you may want to consider using the Application Support class IString, rather than stringstream, to handle strings. The IString class

provides a much broader range of string-handling capabilities than `stringstream`, including the ability to use mathematical operators such as `+` (to concatenate two strings), `=` (to copy one string to another), and `==` (to compare two strings for equality). See Chapter 19, “String Classes” on page 199 for further information.

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

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

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

CLB3ASAD

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

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

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

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

This program produces the following output:

The sum is: 578.12.

Chapter 6. Manipulators

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

Introduction to Manipulators

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

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

Simple Manipulators and Parameterized Manipulators

There are two kinds of manipulators:

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

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

CLB3AMN2

```
// Using simple and parameterized manipulators

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

Creating Simple Manipulators

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

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

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

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

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

This program produces the following output:

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

Creating Simple Manipulators for Your Own Types

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

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

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

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

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

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

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

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

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

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

This statement produces the following output:

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

Creating Parameterized Manipulators for Your Own Types

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

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

```
typedef long double& LONGDBLREF
```
2. Determine the class of your manipulator. If you want to define the manipulator as shown in “Example of Defining an APP Parameterized Manipulator” on page 68, choose a class that has APP in its name (an APP class, also known as an *applicator*). If you want to define the manipulator as shown in “Example of Defining a MANIP Parameterized Manipulator” on page 68, choose a class that has MANIP in its name (a MANIP class). Once you have determined which type of class to use, the particular class that you choose depends on the type of object that the manipulator is going to manipulate. The following table shows the class of objects to be modified, and the corresponding manipulator classes.

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

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

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

4. If you chose one of the APP classes in step 2, define the manipulator as shown in “Example of Defining an APP Parameterized Manipulator” on page 68. If you chose one of the MANIP classes in step 2, define the manipulator as shown in “Example of Defining a MANIP Parameterized Manipulator” on page 68. These two methods produce equivalent manipulators.

Note: Parameterized manipulators defined with `IOMANIP` or `IOAPP` are not associative. This means that you cannot use such manipulators more than once in a single output statement. See “Examples of Nonassociative Parameterized Manipulators” on page 69 for more details.

Example of Defining an APP Parameterized Manipulator

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

CLB3AIOM

```
// Creating and using parameterized manipulators

#include <iomanip.h>

// declare class

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

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

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

IOMANIPdeclare(my_class);

// define a manipulator for the class my_class

OAPP(my_class) pre_print=produce_prefix;

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

This program produces the following output:

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

Example of Defining a MANIP Parameterized Manipulator

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

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

class my_class {
public:
    char * s1;
    const char c;
    unsigned short ctr;
```

```

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

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

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

IOMANIPdeclare(my_class);

// define a manipulator for the class my_class

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

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

```

This example produces the same output as the previous example.

Examples of Nonassociative Parameterized Manipulators

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

CLB3AMN1

```

// Nonassociative parameterized manipulators

#include <iomanip.h>

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

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

ostream_withassign ioswa;

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

```

Creating Parameterized Manipulators

```
//  
// The following statements are equivalent to the previous  
// statement, but they do not cause a compile-time error.  
//  
ioswa << mysetw(3) << i1;  
ioswa << mysetw(5) << i2 << endl;  
}
```

Part 3. The Collection Class Library

Chapter 7. Overview of the Collection Class Library	73
Benefits of the Collection Class Library	73
Concrete Classes Provided by the Library	73
Types of Classes in the Collection Class Library	77
Flat Collections	78
Restricted Access	81
Trees	82
Auxiliary Classes	83
The Overall Implementation Structure	83
Chapter 8. Instantiating and Using the Collection Classes	89
Instantiation and Object Definition	89
Adding, Removing, and Replacing Elements	90
Cursors	93
Iterating over Collections	96
Copying and Referencing Collections	99
Bounded and Unbounded Collections	100
Chapter 9. Element Functions and Key-Type Functions	101
Introduction to Element Functions and Key-Type Functions	101
Using Member Functions	102
Using Separate Functions	103
Using Element Operation Classes	105
Functions for Derived Element Classes	109
Using Smart Pointers	111
Chapter 10. Tailoring a Collection Implementation	121
Introduction	121
Replacing the Default Implementation	121
The Based-On Concept	122
Provided Implementation Variants	122
Features of Provided Implementation Variants	123
Chapter 11. Polymorphism and the Collections	131
Introduction to Polymorphism	131
Using the Abstract Class Hierarchy	131
Adding and Overloading Member Functions	132
Chapter 12. Support for Notifications	135
Chapter 13. Thread Safety and the Collection Classes	139
Guard Objects	139
Restrictions	141
Chapter 14. Exception Handling	143
Introduction to Exception Handling	143
Precondition and Defined Behavior	144
Levels of Exception Checking	145
List of Exceptions	145

The Hierarchy of Exceptions	147
Chapter 15. Collection Class Library Tutorials	149
Preparing for the Lessons	150
Lesson 1: Defining a Simple Collection of Integers	150
Lesson 2: Adding, Listing, and Removing Elements	153
Lesson 3: Changing the Element Type	158
Lesson 4: Changing the Collection	163
Lesson 5: Changing the Implementation Variant	171
Errors When Compiling or Running the Lessons	173
Other Tutorials	173
Chapter 16. Solving Problems in the Collection Class Library	177
Cursor Usage	177
Element Functions and Key-Type Functions	178
Key Access Function - How to Return the Key (1)	179
Key Access Function - How to Return the Key (2)	180
Definition of Key-Type Functions	180
Exception Tracing	181
Declaration of Template Arguments and Element Functions (1)	181
Declaration of Template Arguments and Element Functions (2)	181
Declaration of Template Arguments and Element Functions (3)	182
Default Constructor	182
Chapter 17. Compatibility Information	185
Compatible Items	185
Incompatible Items	186

Chapter 7. Overview of the Collection Class Library

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

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

Benefits of the Collection Class Library

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

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

The Collection Class Library lets you choose the appropriate abstract data type for a given situation by providing *collection classes* that are a complete, systematic, and consistent combination of basic properties. These properties, which are explained in “Flat Collections” on page 78, help you to select abstract data types that are at the appropriate level of abstraction. In a particular application, for example, you may have the choice between using a bag and a key sorted set. The properties of these two collections will help you decide which one is more appropriate.

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

Concrete Classes Provided by the Library

This section lists the concrete collections of the Collection Class Library, and provides a verbal description of a potential application of each collection type. These descriptions are also found in the individual class chapters in the Collection Class Library section of the *OS/390 C/C++ IBM Open Class Library Reference*. You can use these descriptions to understand the characteristics and behavior of each concrete collection, and of the overall capabilities of the Collection Classes.

Bag

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

Sorted Bag

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

Key Bag

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

Key Sorted Bag

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

Set

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

Sorted Set

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

Key Set

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

Key Sorted Set

An example of using a key sorted set is a program that keeps track of canceled credit card numbers and the individuals to whom they are issued. Each card number occurs only once, and the collection is sorted by card number. When a merchant enters a customer's card number into a point-of-sale terminal, the collection is checked to see if that card number is listed in the collection of canceled cards.

Map

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

Sorted Map

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

Relation

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

already contain a family member with that key, before adding the family member.) You can locate a member of the family, but the family members are not in any particular order.

Sorted Relation

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

Sequence

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

Equality Sequence

An example of using an equality sequence is a program that calculates, and places in a collection, members of the *Fibonacci series*, which is a series of integers in which each integer is equal to the sum of the two preceding integers. Multiple elements of the same value are allowed. For example, the sequence begins with two instances of the value 1. Element equality allows you to search for a given element, for example 8, and find out what element follows it in the sequence.

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.

Stack

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

Queue

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

Deque

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

Priority Queue

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

Types of Classes in the Collection Class Library

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

Flat Collections

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

See “Flat Collections” on page 78 for more information on flat collections and their properties.

Trees

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

See “Trees” on page 82 for more details on trees.

Auxiliary Classes

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

Cursors and applicators give you convenient methods for accessing the elements stored in the collections. See “Cursors” on page 93 for more details on cursor classes. See “Iteration Using Applicators” on page 98 for more details on applicator classes.

The pointer classes provide the means to store in collections a pointer to an object instead of the object itself. The managed pointer class offers this object management together with automatic storage

management. See “Using Smart Pointers” on page 111 and “Managed Pointers” on page 116 for more details on pointer classes.

Flat Collections

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

Ordering

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

Access by key

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

Equality for elements

Whether equality is defined for the element.

Uniqueness of entries

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

Figure 8 shows the flat collection that results from each combination of properties. For example, “Map” appears in the Unique, Unordered column for the Key, Element Equality row. This means that a map is unordered, each element is unique, keys are defined, and element equality is defined. This implies that there are no flat collections that have all of the following properties:

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

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

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

Figure 8. Combination of Flat Collection Properties

Ordering of Collection Elements

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

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

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

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

Access by Key

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

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

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

Equality for Keys and Elements

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

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

The equality relation for keys may be different than the equality relation for elements. Consider, for example, a job control block that has a priority and a job identifier that defines equality for jobs. You could choose to implement a job

collection as unordered, with the job ID as key, or as sorted by priority, with the priority as key. The Job class for this job control block could look like this:

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

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

Functions like `locateElementWithKey()` (described in Chapter 15, “Flat Collection Member Functions” in the *OS/390 C/C++ IBM Open Class Library Reference*) use the equality relation on keys to locate elements within a collection. A collection that defines key equality may also define element equality. Functions that are based on equality (such as `locate()`) are only provided for collections that define element equality. Collections that define neither key equality nor element equality, such as heaps and sequences, provide no functions for locating elements by their values or testing for containment. Elements can be added and retrieved from such collections by iteration. For sequences, elements can also be added and retrieved by position.

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

Keys can be used to access a particular element in a collection. The alternative to defining element equality as equality of all data members is to define it as equality of keys only. (In the job control block example on page 79, this means defining job equality as equality of the job ID.) Use this alternative only when you are sure that keys are unique. When you use this alternative, you can locate an element only with the key (using `locateElementWithKey(key)` instead of `locate(element)`). Locating elements by key improves performance, particularly if the complete element is large or difficult to construct in comparison to the key alone. Consider the two alternatives in the following example:

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

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

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

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

Uniqueness of Entries

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

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

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

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

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

Restricted Access

Flat collections with restricted access have a restricted set of functions that can be applied to them; that is, only a subset of the functions listed in Chapter 15, "Flat Collection Member Functions" in the *OS/390 C/C++ IBM Open Class Library Reference* can be applied. Examples of such flat collections are stack and priority queue.

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

1. You can simplify the interface to the collection.

Trees

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

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

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

See Part 3, “Flat Collection Classes” in the *OS/390 C/C++ IBM Open Class Library Reference* for descriptions of collections with restricted access. These descriptions are alphabetically merged with descriptions for other collections. You can use Table 4 to select the appropriate flat collection with restricted access for a given set of properties.

Table 4. Properties for Collections with Restricted Access

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

Trees

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

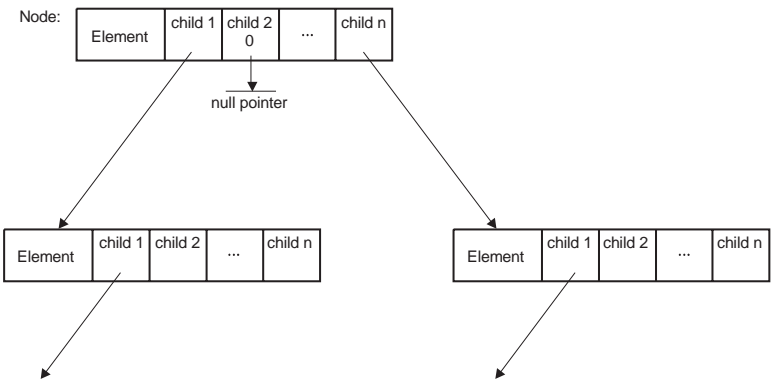


Figure 9. The Structure of N-ary Trees

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

Generally, you can locate and insert elements in collections implemented as trees faster than you can in collections implemented as lists. However, if you only want to iterate through elements in a collection, it is faster to iterate through the elements of a collection if it is implemented as a list.

Auxiliary Classes

Auxiliary classes are those classes that support other classes, and include classes for cursors, pointers and iterators. To use the collection classes, you also need a *cursor* class for referencing an element in a collection, and an *applicator* class for iterating over a collection. These are described in “Cursors” on page 93 and “Iteration Using Applicators” on page 98.

You can use the *smart pointer* classes to manage objects; they enable automatic storage management. “Using Smart Pointers” on page 111 and “Managed Pointers” on page 116 explain the concepts and usage in detail.

The Overall Implementation Structure

To achieve maximum runtime efficiency and ease of use, the Collection Class Library combines the common features of object-oriented techniques, such as class hierarchies and polymorphism, with an efficient class structure that uses advanced optimization techniques. This section gives a brief overview of the Collection Class structure that is shown in Figure 10 on page 84. A more detailed explanation of the particular concepts is found in subsequent sections.

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

Use the Defaults

Default implementations are provided for every collection. If you do not want to be concerned with choosing an implementation for an abstract data type, you can use the *default classes* provided by the Collection Classes. In chapters of the *OS/390 C/C++ IBM Open Class Library Reference* that describe particular collections, the default implementation is the first implementation in the “Class Implementation Variants” table for that chapter, if a table is present. If no table is present, the default implementation is stated in the chapter’s “Class Implementation Variants” section.

Use Variants

If you want to choose a particular implementation variant for a collection, you can easily replace the default implementation by an implementation variant of the same collection that behaves externally in the same way but may offer improved performance for your concrete application, depending on its characteristics.

Use Polymorphism and Abstract Classes

If you want to have a more generalized collection class than those offered by the concrete classes, you can take advantage of polymorphism. For example, when working with a set, instead of using the concrete classes *ISet*, *IGSet*, *ISetAsBstTree*, and so on, you can use the abstract class *IASet* or, for more generic behavior, the abstract class *IAEqualityCollection*. *Abstract classes* let you program to a more generalized interface, without necessarily knowing what

abstract data types (collections), your code will operate on. You can leave the implementation details for later.

Categories of Classes

Figure 10 illustrates the relationships between the categories of classes for the collection known as a set. Each class falls within one of the following categories: concrete, typed implementation, typeless implementation, and abstract classes. Arrows indicate a relationship between classes. The relationships are:

- Instantiates (arrow with dashed line)
- Is a (line with triangle)
- Has a (arrow with solid line)

In this figure, you will notice certain naming conventions. For example, default classes begin with the letter I, while abstract classes begin with the letters IA. For information on naming conventions, see “Class Template Naming Conventions” on page 86.

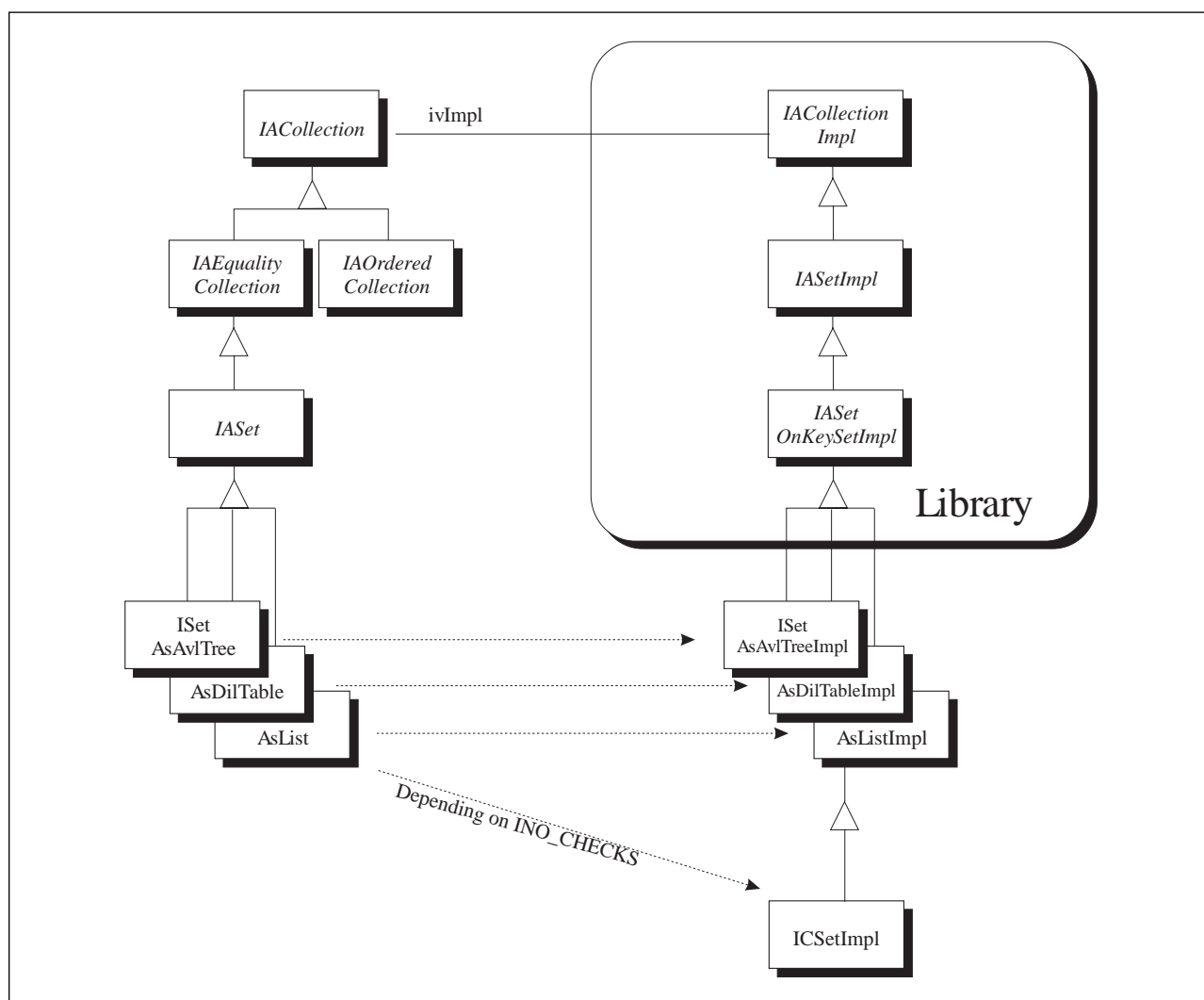


Figure 10. Overall Structure of Collection Classes

The class ISet uses an AVL tree as the default implementation. The other implementation variants are linked list and diluted table. The three implementation

variants `ISetAsAvlTree`, `ISetAsList`, and `ISetAsDilTable` are subclasses of `IASet`. If you do not want to deal with implementation variants, you can just use the default class `ISet`. Additional information is found in Chapter 10, “Tailoring a Collection Implementation” on page 121.

The following sections describe the categories of Collection Classes.

Default Classes

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

- A class that is instantiated only with the element type, and possibly the key type. `ISet` is an example of this type of default class.
- A class that takes element-specific functions. `IGSet` is an example of this type of default class. See “Using Element Operation Classes” on page 105 for information on element-specific functions.

Variant Classes

Each abstract data type can be instantiated either by its default class or by one of several variant classes. Sets can be implemented, for example, as AVL trees, lists, or hash tables. Default classes and variant classes are also called the implementation variants of a collection. All implementation variants of a collection have the same interface and external behavior.

Collection Class Hierarchy

The classes in the Collection Classes are all related through the hierarchy of abstract classes shown in Figure 11 on page 86. An *abstract class* is 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, and classes derived from it represent implementations of the concept. You cannot construct an object of an abstract class. With abstract classes, you can program to a more generalized interface without knowing what abstract data types, or collections, the code will operate on. Implementation details can be left for later.

In the figure, abstract classes have a grey shadow. Concrete collections have a black shadow, or a white shadow for restricted access collections. The leaves of the abstract class hierarchy (that is, those classes that have no derived classes within the abstract class hierarchy tree) define the collection for which concrete implementations are provided. The lines in the figure represent an *is a* relationship from a lower collection to the collection above it. For example, a set is an equality collection, which is a collection. The names of abstract collections start with `IA`. See Chapter 11, “Polymorphism and the Collections” on page 131 for more details on the use of polymorphism in the Collection Classes.

Typed and Typeless Implementation Classes

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

Typeless implementation classes prevent unnecessary code expansion, which could occur if all code for a collection were fully implemented through its templates. For example, the `add(Element const& element)` function is offered with a typed interface, so that the compiler can check whether a program tries to add a string to

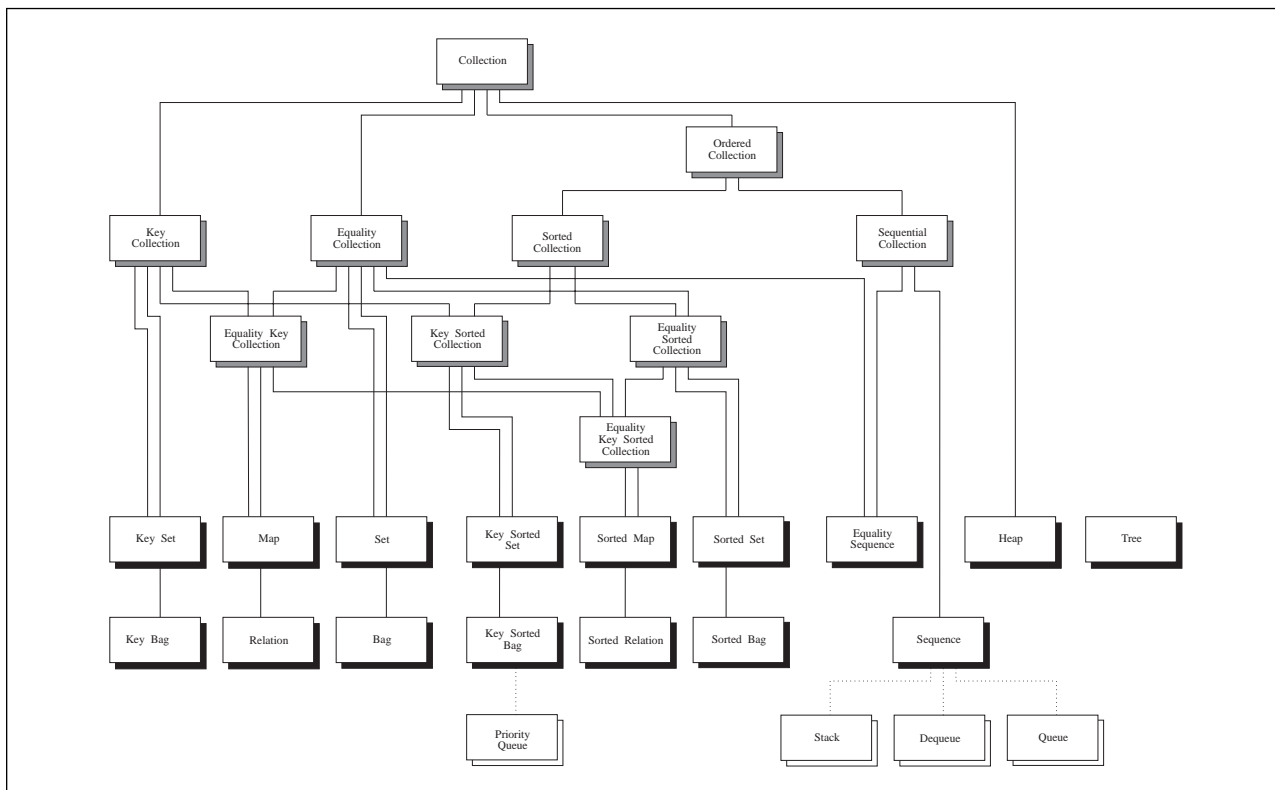


Figure 11. The Collection Class Hierarchy. Abstract classes have a grey background. Concrete classes have a black background. Restricted access classes have a white background. Dotted lines show a “based-on” relationship, not an actual derivation.

a collection of integers. However, suppose an application were to use all of the following:

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

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

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

Class Template Naming Conventions

All class templates begin with an uppercase I. Table 5 on page 87 shows the naming conventions used to distinguish between different types of class templates, given a default class template of `ISet`. Underscored letters in each class template name are those that indicate the stated convention:

Class name	Meaning of letters
<code>ISet</code>	Default class template.
<code>ISet<u>I</u>mpl</code>	Typeless implementation class.
<code>IC<u>S</u>etImpl</code>	Typeless implementation class that implements additional checks.
<code>IG<u>S</u>et</code>	Default generic class template. The element operations class can be specified as template argument.
<code>ISet<u>A</u>sAvlTree</code> <code>ISet<u>A</u>sBstTree</code> <code>ISet<u>A</u>sList</code> <code>ISet<u>A</u>sTable</code> <code>ISet<u>A</u>sDillTable</code> <code>ISet<u>A</u>sHshTable</code>	Variant class templates.
<code>IA<u>S</u>et</code>	Abstract class template.
<code>IY<u>S</u>et</code>	Default notification-enabled class template.

Table 5. Class Template Naming Conventions

Chapter 8. Instantiating and Using the Collection Classes

This chapter describes how to instantiate and use collection classes. To use a collection class, you normally follow these three steps:

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

Instantiation and Object Definition

This section describes instantiation for the default implementation. Consider the following example header file for a class Person:

```
//person.h - Header file containing class Person
#include <iostream.h>
#include <string.hpp>

class Person {
    IString PersonName;
    IString TNumber;

public:
    //constructor
    Person ():PersonName(""),TNumber("") {};
    //copy constructor
    Person(IString Name,IStringNumber):PersonName(Name),TNumber(Number)
    {};

    IString const& GetPersonName() const {return PersonName;};
    IString const& GetTNumber() const {return TNumber;};
    IBoolean operator== (Person const& A) const {
        return (PersonName==A.GetPersonName()) &&
            (TNumber==A.GetTNumber());};
    IBoolean operator< (Person const& A) const {
        return (PersonName < A.GetPersonName());};
};
```

For a given class, such as ISet, and a given element type, such as a class named Person, the instantiation for a new class that represents sets of persons could look like this:

```
//main.cpp - main file
#include <iset.h>
#include <iostream.h>
#include "person.h" //person.h from the previous example

typedef ISet <Person> AddressList;

void main() {
    AddressList Business;
    Person A("Peter Black","50706");
    Business.add(A);
    cout << "\nThe set now contains " << Business.numberOfElements() << " entries!\n";
}
```

Once the AddressList collection is defined, you can define AddressList objects Family, Business, and Sportclub as follows:

```
AddressList Family, Business, Sportclub;
```

Adding, Removing, and Replacing Elements

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

```
ISet < Person > Family, Business, Sportclub;
```

However, you should begin by explicitly defining a named class, such as AddressList, that uses the default implementation. It is then easier to replace the default implementation with a better implementation later on. See Chapter 10, “Tailoring a Collection Implementation” on page 121 for more details on replacing default implementations.

Adding, Removing, and Replacing Elements

You can perform three operations to modify a collection:

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

Adding Elements

The function add() places the element identified by its argument into the collection. It has two general properties:

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

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

Figure 12 on page 91 shows the result of adding a series of four elements to a map, a relation, a key set, and a key bag. The first row shows what each collection looks like after the element <a,1> has been added to each collection. Each following row shows what the collections look like after the element in the leftmost column is added to each.

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

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

Figure 12. Behavior of add for Unique and Multiple Collections

add() behaves differently depending on the properties of the collection:

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

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

```
//main.cpp - main file
#include <iset.h>
#include <iostream.h>
#include "person.h" //person.h from the previous examples

typedef ISet <Person> AddressList;

void main() {
    AddressList Business;
    Person A("Peter Black","714-50706");
    Person B("Carl Render","714-540321");
    Person C("Sandra Summers","214-660012");
    Business.add(A);
    Business.add(B);
    Business.add(C);
    Business.add(A); //Person A is added for the second time
    cout << "\nThe set now contains " << Business.numberOfElements()
         << " entries!\n";
}
/* If you run the program, the set will only contain 3 different
   entries. In a set, each element is unique. No two elements
   can be the same. To illustrate the difference between sets and
   bags, run the program using a bag rather than a set. */
```

Removing Elements

In the Collection Classes, you can remove an element that is pointed to by a given cursor by using the removeAt() function. All other removal functions operate on the model of first generating a cursor that refers to the desired position and then removing the element to which the cursor refers. Additional information about cursors is found in “Cursors” on page 93. There is an important difference

Adding, Removing, and Replacing Elements

between element *values* and element *occurrences*. An element value may, for nonunique collections, occur more than once. The basic `remove()` function always removes only one occurrence of an element.

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

After you have removed one element with the property, the entire collection would have to be searched for the next element with the property. If you want to remove all of the elements in a collection that have a given property, you should use the function `removeAll()` and provide a *predicate* function as its argument. This predicate function has an element as argument and returns an `IBoollean` value. The `IBoollean` result tells whether the element will be removed. Consider the following example:

```
//main.cpp - main file
#include <iset.h>
#include <iostream.h>
#include "person.h" //person.h from the previous examples

typedef ISet <Person> AddressList;

IBoollean noPhone(Person const& P,void*) //predicate function
{
    return P.GetTNumber()=="x";
}

void main() {
    AddressList Business;
    Person A("Peter Black","714-50706");
    Person B("Carl Render","714-540321");
    Person C("Sandra Summers","x");
    Person D("Mike Summers","x");
    Business.add(A);
    Business.add(B);
    Business.add(C);
    Business.add(D);
    Business.add(A); //Person A is added for the second time
    cout << "\nThe set now contains " << Business.numberOfElements()
         << " entries!\n";
    Business.removeAll (noPhone); //Person B is removed from the set
    cout << "\nThe set now contains " << Business.numberOfElements()
         << " entries!\n";
}

/* If you run the program, the set will only contain 2 elements
as a result of the the remove function. Try modifying the
program so that all persons with a telephone number are
removed when the program is run. */
```

Sometimes you may want to pass more information to the predicate function. You can use an additional argument of type `void*`. The pointer then can be used to access a structure containing further information. See the last example under “Iteration Using `allElementsDo`” on page 97 for information on how to use the additional argument.

Replacing Elements

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

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

Sequential collections and heaps do not have a positioning property. Element values in sequences and heaps can be changed freely. Replacing element values involves copying the whole value. If only a small part of the element is to be changed, it is more efficient to use the `elementAt()` access function described in “Using Cursors for Locating and Accessing Elements” on page 94. The `replaceAt()` function checks whether the replacing element has the same positioning property as the replaced element. (See Chapter 14, “Exception Handling” on page 143 for more details on preconditions.) When you use the `elementAt()` function to replace part of the element value, this check is not performed. If you want to ensure safe replacement (a replacement that does not change the positioning property), use `replaceAt()` rather than `elementAt()`.

Cursors

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

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

```
//main.cpp - main file
#include <iset.h>
#include <iostream.h>
#include "person.h" //person.h from the previous examples

typedef ISet <Person> AddressList;

void main() {
    AddressList Business;
    AddressList::Cursor myCursor(Business); //Cursor definition
    Person A("Peter Black","714-50706");
    Person B("Carl Render","714-540321");
    Person C("Sandra Summers","x");
    Person D("Mike Summers","x");
    Business.add(A);
    Business.add(B);
    Business.add(C);
    Business.add(D);
    Business.add(A); //Person A is added for the second time
    cout << "\nThe set now contains " << Business.numberOfElements()
```

```
        <<" entries!\n";  
    }
```

The following two lines of code are functionally equivalent:

```
myCursor.setToNext();  
Business.setToNext(myCursor);
```

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

- An iteration scheme that is simpler than using applicators. (See “Iteration Using allElementsDo” on page 97.)
- The ability to define functions that return cursors. Such functions can give you fast access to an element if it exists, or indicate the non-existence of an element by returning an invalid cursor.

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

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

Because all cursors of the collection become undefined when elements are removed, removing all elements with a given property from a collection cannot be done efficiently using cursors.

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

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

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

Using Cursors for Locating and Accessing Elements

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

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

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

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

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

```
//main.cpp - main file
#include <iset.h>
#include <iostream.h>
#include "person.h" //person.h from the previous examples

typedef ISet <Person> AddressList;

void main() {
    AddressList Business;
    AddressList::Cursor myCursor(Business); //Cursor definition
    Person A("Peter Black","714-50706");
    Person B("Carl Render","714-540321");
    Person C("Sandra Summers","x");
    Person D("Mike Summers","x");
    Person E;
    Business.add(A);
    Business.add(B);
    Business.add(C);
    Business.add(D);
    if (Business.locate(B,myCursor)){
        E=Business.elementAt(myCursor) ;
    } else {
        cout << "\nElement not in set !";
    } /* endif */
    Business.remove(B); //myCursor is no longer valid
    if (Business.locate(B,myCursor)) {
        E=Business.elementAt(myCursor);
    } else {
        cout << "\nElement not in set !";
    } /* endif */
}
```

The `elementAt()` function can also be used to replace the value of the referenced element. You must ensure that the positioning property of the element is not changed with respect to the given collection. See “Adding, Removing, and Replacing Elements” on page 90 for more details.

There are two versions of `elementAt()`:

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

Use the first version of `elementAt()` if you want to ensure that the located element cannot be changed by any subsequent function.

Iterating over Collections

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

- Using cursors
- Using the `allElementsDo()` function together with applicators or applicator functions

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

You cannot add or remove elements from a collection while you are iterating over a collection, or all elements may not be visited once. You cannot use any of the iterations described in this section if you want to remove all of the elements of a collection that have a certain property. Use the function `removeAll()` (described in Chapter 15, "Flat Collection Member Functions" in the *OS/390 C/C++ IBM Open Class Library Reference*), that takes a predicate function as argument. See "Removing Elements" on page 91 for details on removing elements.

Iteration Using Cursors

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

```
//main.cpp - main file
#include <iset.h>
#include <iostream.h>
#include "person.h" //person.h from the previous examples

typedef ISet <Person> AddressList;

ostream& operator<<(ostream& os, Person A) {
    return (os << endl << A.GetPersonName() << " "<< A.GetTNumber());
}

void main() {
    AddressList Business;
    AddressList::Cursor myCursor(Business);
    Person A("Peter Black", "714-50706");
    Person B("Carl Render", "714-540321");
    Person C("Sandra Summers", "x");
    Person D("Mike Summers", "x");
    Person E;
    Business.add(A);
    Business.add(B);
    Business.add(C);
    Business.add(D);

    //List of all elements in the set
    for (myCursor.setToFirst(); myCursor.isValid(); myCursor.setToNext())
    {
        cout << Business.elementAt(myCursor);
    }
}
```

`AddressList::Cursor` is the class `Cursor` that is defined within the class `AddressList`. This is referred to as a *nested class*. `myCursor` is the name of the cursor object. Its constructor takes `Business` as an argument.

The Collection Classes define a macro `forICursor` that lets you write a cursor iteration even more elegantly:

```

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

forICursor(myCursor)
{
    cout << Business.elementAt(myCursor);
}

```

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

```

forICursor(myCursor)
{
    cout << myCursor.element(); //myCursor is associated to Business
}

```

The function `element()` above is a function of the `Cursor` class (see “Cursors” on page 93). It returns a **const** reference to the element currently pointed at by the cursor. The element returned might therefore not be modified. Otherwise it would be possible to manipulate a constant collection by using cursors.

Note: You should remove multiple elements from a collection using the `removeAll()` function, with a predicate function as an argument. See “Adding, Removing, and Replacing Elements” on page 90 for further details.

Iteration Using `allElementsDo`

Cursor iteration has two possible drawbacks:

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

The Collection Classes provide the `allElementsDo()` function that addresses both drawbacks by calling a function that is applied to all elements.

The function returns an `IBoollean` value that tells whether the iteration should be continued or not. For ordered collections, the function is applied in this order. Otherwise the order is unspecified.

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

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

`allElementsDo(myApplicatorObject)`, where *myApplicatorObject* is an object of `MyApplicatorClass`.

Iteration Using Applicators

Defining a function as a member function of a user-defined applicator class provides more flexibility than coding the function that you want to be applied to all elements as a C++ function. You can better encapsulate the member function, and you can use additional arguments to that function if needed. If the function is a method that you can use for various classes, you can reuse the applicator class.

By definition, an applicator is an object created from a class that is derived from `IApplicator` or `IConstantApplicator`. This applicator class contains the member function `applyTo`, which is applied to every element in a collection using the `allElementsDo` function. This member function returns an `IBoolean` value that indicates whether an iteration should be continued or not.

Note: You should not add or remove elements while using the applicator.

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

```
template < class Element, ... >
class ISet {
    // ...

    // Iteration applying a C++ function:
    IBoolean allElementsDo (IBoolean (*function)(Element&, void*),
                           void* additionalArgument = 0);

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

    // Iteration applying an applicator object:
    IBoolean allElementsDo (IApplicator < Element > &);
    IBoolean allElementsDo (IConstantApplicator < Element > &)const;
};
```

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

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

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

Additional arguments that are needed for the iteration can, for example, be passed as arguments to the constructor of the derived applicator class. You must define

the function with the given argument and return types. For additional arguments, you may have to define a separate class or structure.

The following example shows the use of applicators.

```
// SUMUP - An example of using applicators
//main.cpp - main file
#include <iset.h>
#include <iostream.h>
#include "person.h" //person.h from the previous examples

typedef ISet <Person> AddressList;

ostream& operator<<(ostream& os, Person A) {
    return (os << endl << A.GetPersonName() << " "<<A.GetTNumber());
}

class ListApplicator: public IConstantApplicator <Person> {
public:
    IBoolean applyTo (Person const& A) {
        cout << A;
        return true;
    }
};

void ListFunction (AddressList const& List) {
    ListApplicator LA;
    List.allElementsDo (LA);
}

void main() {
    AddressList Business;
    AddressList::Cursor myCursor(Business);
    Person A("Peter Black", "714-50706");
    Person B("Carl Render", "714-540321");
    Person C("Sandra Summers", "x");
    Person D("Mike Summers", "x");
    Person E;
    Business.add(A);
    Business.add(B);
    Business.add(C);
    Business.add(D);

    //List of all elements in the set
    ListFunction(Business);
}

/* This time you get the address listing using an applicator */
```

Copying and Referencing Collections

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

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

```
void removeListMember (AddressList aList) { /* ... */ } // wrong
void removeListMember (AddressList &aList) { /* ... */ } // right
```

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

Bounded and Unbounded Collections

```
AddressList f() {  
    AddressList aList;  
    // ...  
    return aList;  
}  
Business=f(); //Very inefficient
```

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

```
void f (AddressList & aList) { /* ... */ }  
// ...  
f(Business);
```

Bounded and Unbounded Collections

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

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

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

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

Chapter 9. Element Functions and Key-Type Functions

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

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

Introduction to Element Functions and Key-Type Functions

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

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

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

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

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

- Equality test
- Ordering relation
- Hash function

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

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

- Allocation
- Deallocation

The lists above are the superset of all element functions and key-type functions that a Collection Class can ever require. For example, a collection without keys does not require any key-type functions, and a collection without element equality does not require an equality test. Element functions and key-type functions required for

a certain collection are listed with the description of each collection in the *OS/390 C/C++ IBM Open Class Library Reference*.

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

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

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

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

Using Member Functions

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

You must define these functions using member functions:

- Constructors
- Destructors

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

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

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

The Collection Class Library does not check or use the return type of `operator=()`. The return type of equality and ordering relation must be compatible with type `IBoolean`.

Using Separate Functions

You can use separate functions to provide the required element and key functions. A separate function is a function that is not a member of any class. Use separate functions when, in instantiating the Collection Class, you have no control over the element class, and the element class does not define the appropriate functions.

The following functions must be defined as separate functions that are not members of any class:

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

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

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

You can find examples of these functions in the tutorials (see Chapter 15, “Collection Class Library Tutorials” on page 149) and in the coding examples in the *OS/390 C/C++ IBM Open Class Library Reference*.

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

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

The `compare()` function must return a value that is less than, equal to, or greater than zero, depending on whether the first argument is less than, equal to, or greater than the second argument.

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

Note: An efficient hash function is very important to the performance of your program. If you are unsure of how to implement an efficient hash function, see the suggested reading material on data structures and algorithms in “Suggested Reading” on page xl.

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

Using Separate Functions

integer types because the result of a-b can be used to determine whether a<b evaluates to true.)

The following examples demonstrate the use of a separate function for the definition of the key access. The element class is Person, its data member PersonName is the key, and its member function GetPersonName() is used to access the key. The example below is the header file:

```
//person.h - header file containing class Person
#include <iostream.h>
#include <string.hpp>

class Person {
    IString PersonName;          //This will be used as the key
    IString TNumber;

public:
    //constructor
    Person ():PersonName(""),TNumber("") {};
    //copy constructor
    Person(IString Name,IString Number):PersonName(Name),TNumber(Number)
    {};

    IString const& GetPersonName() const {return PersonName;};
    IString const& GetTNumber() const {return TNumber;};
    IBoolean operator== (Person const& A) const {
        return (PersonName==A.GetPersonName()) &&
            (TNumber==A.GetTNumber());
    };
    IBoolean operator< (Person const& A) const {
        return (PersonName < A.GetPersonName());
    };
};

ostream& operator<<(ostream& os,Person A);

// Use separate function Key const& key (Element const&);

inline IString const& key (Person const& A) //Key access
{ return A.GetPersonName();};
```

The example below is the main file.

```
//main.cpp - main file
#include "person.h" //person.h from the previous example
#include <ikeyset.h>

typedef IKeySet <Person,IString> AddressList;

ostream& operator<<(ostream& os,Person A) {
    return (os << endl << A.GetPersonName() << " "<<A.GetTNumber());
}

void main() {
    AddressList Business;
    AddressList::Cursor myCursor(Business);
    Person A("Peter Black","714-50706");
    Person B("Carl Render","714-540321");
    Person C("Sandra Summers","x");
    Person D("Mike Summers","x");
    Business.add(A);
    Business.add(B);
    Business.add(C);
    Business.add(D);
    Business.removeElementWithKey("Carl Render");
    forICursor(myCursor) {
        cout<<Business.elementAt(myCursor);
    }
}
```

Using Element Operation Classes

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

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

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

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

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

For an example of using element operation classes, see “Coding Example for Map” in the *OS/390 C/C++ IBM Open Class Library Reference*.

The following excerpt shows the definition of the class templates for `ISequenceAsList` and `IGSequenceAsList`:

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

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

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

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

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

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

Using Element Operation Classes

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

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

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

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

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

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

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

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

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

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

To define an operations class, use the predefined templates for standard functions, and define the specific functions individually. Consider, for example, persons that have a name (PersonName) and a phone number (TNumber). The name serves as the key for an address list, while the phone number serves as the key for a phone list. Because the key() function is already defined to yield the person name, the phone list has to be instantiated in the following way:

```
//person.h - header file containing class Person
#include <iostream.h>
#include <istring.hpp>
#include <istdops.h>

class Person {
    IString PersonName;
    IString TNumber;

public:
    //constructor
    Person ():PersonName(""),TNumber("") {};
    //copy constructor
    Person (IString Name,IString Number):PersonName(Name),TNumber(Number)
        {};

    IString const& GetPersonName() const {return PersonName;};
    IString const& GetTNumber() const {return TNumber;};
    IBoolean operator== (Person const& A) const {
        return (PersonName==A.GetPersonName()) &&
            (TNumber==A.GetTNumber());
    };
    IBoolean operator< (Person const& A) const {
        return (PersonName < A.GetPersonName());
    };
};

ostream& operator<<(ostream& os,Person A);

class PhoneOps:public IStdMemOps,public IStdAsOps<Person> {
public:
    IString const& key (Person const& A) const {return A.GetTNumber();}
    IStdCmpOps <IString> keyOps;
};

inline IString const& key (Person const& A)    //Key access
{ return A.GetPersonName();};
```

The following example is the main file:

```
//main.cpp - main file
#include "person.h" //person.h from the previous example
#include <istdops.h>
#include <ikeyset.h>

typedef IKeySet <Person,IString> AddressList;
typedef IKeySet <Person,IString,PhoneOps> PhoneList;

ostream& operator<<(ostream& os,Person A) {
    return (os << endl << A.GetPersonName() <<" "<<A.GetTNumber());
}

void main() {
    AddressList Business;
    PhoneList PhoneBook;

    AddressList::Cursor myCursor1(Business);
    PhoneList::Cursor myCursor2(PhoneBook);

    Person A("Peter Black","714-50706");
    Person B("Carl Render","714-540321");
    Person C("Sandra Summers","x");
    Person D("Mike Summers","x");
    Business.add(A);
    Business.add(B);
    Business.add(C);
    Business.add(D);
    PhoneBook.add(A);
    PhoneBook.add(B);
    PhoneBook.add(C);
    PhoneBook.add(D);

    cout << "\n\nPhoneBook before removing an element: ";
    forICursor(myCursor2) {
        cout<<PhoneBook.elementAt(myCursor2);
    }

    cout << "\n\nPhoneBook after removing an element: ";
    PhoneBook.removeElementWithKey("714-50706");
    forICursor(myCursor2) {
        cout<<PhoneBook.elementAt(myCursor2);
    }

    cout << "\n\nBusiness before removing an element: ";
    forICursor(myCursor1) {
        cout<<Business.elementAt(myCursor1);
    }

    cout << "\n\nBusiness after removing an element: ";
    Business.removeElementWithKey("Peter Black");
    forICursor(myCursor1) {
        cout<<Business.elementAt(myCursor1);
    }
}

/* Questions: Why does the PhoneBook only contain 3 elements?
              Why are both lists in a different order? */
```

The functions that are required for a particular Collection Class depend not only on the abstract class but also on the concrete implementation choice. If you choose a set to be implemented through a hash table, the elements require a hash function. If you choose a (sorted) AVL tree implementation, elements need a comparison function. Even the default implementations may require more functions to be provided than would be necessary for the collection interface. Each chapter in the *OS/390 C/C++ IBM Open Class Library Reference* that describes a particular collection defines which functions must be provided for keys and elements for each implementation of that collection.

Memory Management with Element Operation Classes

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

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

```
IGSequenceAsList<Element, MyOperationsClass>
```

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

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

Functions for Derived Element Classes

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

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

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

The classes `IElemPointer`, `IMngElemPointer`, and `IAutoElemPointer` (see “Managed Pointers” on page 116) internally use a function called `elementForOps()` to direct functions such as `equal()` and `compare()` to the referenced element, so that they are not applied to the pointer itself and so that instantiations such as `ISet <IElemPointer <Person>>` perform the functions on the elements. This indirection is usually transparent but you must consider it when you derive classes from the `IElemPointer` class. The standard operation classes first apply a function `elementForOps()` to the element before they apply the corresponding non-member (`equal()`, ...) function. By default, a corresponding template function is instantiated for `elementForOps()` which takes an element as input and returns that element. For pointer classes that perform operations on the pointers themselves (`IAutoPointer`, `IMngPointer`), this function takes the pointer as input and returns the same pointer.

Functions for Derived Element Classes

For pointer classes that perform the operations on the referenced elements (IElemPointer, IAutoElemPointer, IMngElemPointer), this function takes the pointer as input and returns the referenced element. If a class derived from IElemPointer<E> is used as a collection element type, the default template functions must be instantiated before a conversion will be considered. A derived class must therefore explicitly redefine the elementForOps() function, as shown in the following example, where class PersonPtr redefines both versions of elementForOps() by calling the default elementForOps() with a PersonPtr as argument. Both versions are then made to return a cast to Person reference. Consider the following header file example:

```
//person.h - header file containing class Person
#include <iostream.h>
#include <istring.hpp>
#include <iptr.h>

class Person {
    IString PersonName;          //This will be used as the key
    IString TNumber;

public:
    //constructor
    Person ():PersonName(""),TNumber("") {};
    //copy constructor
    Person(IString Name,IString Number):PersonName(Name),TNumber(Number)
    {};

    IString const& GetPersonName() const {return PersonName;};
    IString const& GetTNumber() const {return TNumber;};
    IBoolean operator== (Person const& A) const {
        return (PersonName==A.GetPersonName()) &&
            (TNumber==A.GetTNumber());
    };
    IBoolean operator< (Person const& A) const {
        return (PersonName < A.GetPersonName());
    };
};

class PersonPtr: public IElemPointer <Person> {

    friend inline Person& elementForOps (PersonPtr& A) {
        return (Person&) elementForOps ((IElemPointer<Person> &)A); }

    friend inline Person const& elementForOps (PersonPtr const& A) {
        return (Person const&) elementForOps ((IElemPointer<Person> &) A);}

public:
    PersonPtr(): IElemPointer<Person>() {}
    PersonPtr(Person* ptr,IExplicitInit IINIT)
        :IElemPointer<Person>(ptr,IINIT) {}
};

ostream& operator<<(ostream& os,Person A);

inline IString const& key (Person const& A)    //Key access
{ return A.GetPersonName();};
```

The following example shows the main file.

```
//main.cpp - main file
#include "person.h" //person.h from the previous example
#include <istdops.h>
#include <iset.h>

typedef ISet <PersonPtr> AddressList;

ostream& operator<<(ostream& os,Person A) {
    return (os << endl << A.GetPersonName() << " "<<A.GetTNumber());}
```



```

}

void main() {
    AddressList Business;
    AddressList::Cursor myCursor1(Business);

    PersonPtr Aptr (new Person("Peter Black","714-50706"),IINIT);
    PersonPtr Bptr (new Person("Carl Render","714-540321"),IINIT);
    PersonPtr Cptr (new Person("Sandra Summers","x"),IINIT);
    PersonPtr Dptr (new Person("Mike Summers","x"),IINIT);
    PersonPtr CopyCptr (new Person("Sandra Summers","x"),IINIT);

    Business.add(Aptr);
    Business.add(Bptr);
    Business.add(Cptr);
    Business.add(Dptr);
    Business.add(CopyCptr);

    forICursor (myCursor1) {
        cout << *Business.elementAt(myCursor1);
    }
}
/* Comment: CopyCptr and Cptr refer to different memory addresses, so
both of them could be put into the set. Using ElementPointers
rather than regular pointers, all collection functions are done
on the elements to which the pointers point. That is why a pointer
pointing on Sandra Summers is only entered once into the list. */

```

Using Smart Pointers

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

The Collection Classes define five pointer classes:

- IElemPointer
- IAutoPointer
- IAutoElemPointer
- IMngPointer
- IMngElemPointer

These types are referred to as *smart pointers*. Their main characteristics are:

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

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

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

Overview of Smart Pointers

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

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

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

Pointers from the three `I...Elem...` classes are also called *element pointers*. Element pointers are only useful when you store them in a collection. The elements themselves are not “stored” in the collection, although information from the elements is used by Collection Classes functions. See “Element Pointers” on page 113 for more information on the element pointer types.

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

- `IMngPointer`
- `IAutoPointer`

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

Most smart pointers perform automatic storage deallocation for objects that are no longer referenced. They are:

- `IAutoPointer`
- `IAutoElemPointer`
- `IMngPointer`
- `IMngElemPointer`

Pointers of classes `IAuto...` are called *automatic pointers*. Automatic pointers perform memory management in such a way that referenced objects are deleted as soon as the pointer passes out of scope. See “Automatic Pointers” on page 116 for more information on automatic pointers.

Pointers of classes `IMng...` are called *managed pointers*. Managed pointers perform memory management in such a way that the references to objects are counted, and objects are deleted only when they are no longer referenced by any

managed pointer. See “Managed Pointers” on page 116 for more information on managed pointers.

To exploit the advantage of memory management, you can use non-element pointers (for example, `IMngPointer`) instead of standard C++ pointers without storing the pointers in a collection.

Automatic storage management is particularly useful when functions return pointers or references to objects that they have created (dynamically allocated), and the last user of the object is responsible for cleaning up.

The following features of Collection Classes pointer types give you the choices shown in the table below. Standard C++ pointers are included for comparison.

- Element functions performed on referenced elements
- Element functions performed on pointers
- Automatic storage management

	Destruction of Pointed Objects		
	Not managed	When out-of-scope	Reference counted
Collections call element operations on pointer	Standard C++ pointer	<code>IAutoPointer</code>	<code>IMngPointer</code>
Collections call element operations on referenced object	<code>IElemPointer</code>	<code>IAutoElemPointer</code>	<code>IMngElemPointer</code>

Smart pointers can only take arguments of type `class` or `struct`. This is because the overloaded operator-`>` needs to return an object of such a type. You can apply pointer objects from these five classes in the same way you use ordinary C++ pointers, with the `*` and `->` operators. Elements are implicitly deleted except in the case of `IElemPointer`. To delete an element referred to by an `IElemPointer` you must use an explicit conversion to the referenced element type:

```
IElemPointer < E > ptr;
// ...
delete (E*) ptr;
```

Element Pointers

If you create a collection of C++ pointers or pointers of type `IMngPointer` or `IAutoPointer`, Collection Classes methods that use element comparison functions will do the comparison on the elements' pointers instead of on the elements themselves.

If you do want element functions to work on the pointers instead of the referenced elements, you do not need to implement equality and ordering relation for the chosen pointer type (`IAutoPointer`, `IMngPointer` or C++ pointers). The compiler can instantiate the default element function templates in such cases. If necessary, you can implement your element functions for the referenced element type.

In the following examples, adding, locating, and other functions are based on pointer equality and ordering, and not on the equality defined for the `Person` type. The header file appears below:

Using Smart Pointers

```
//person.h - header file containing class Person
#include <iostream.h>
#include <istring.hpp>
#include <iptr.h>

class Person {
    IString PersonName;          //This will be used as the key
    IString TNumber;

public:
    //constructor
    Person ():PersonName(""),TNumber("") {};
    //copy constructor
    Person(IString Name,IString Number):PersonName(Name),TNumber(Number)
    {};

    IString const& GetPersonName() const {return PersonName;};
    IString const& GetTNumber() const {return TNumber;};
    IBoolean operator== (Person const& A) const {
        return (PersonName==A.GetPersonName()) &&
            (TNumber==A.GetTNumber());
    };
    IBoolean operator< (Person const& A) const {
        return (PersonName < A.GetPersonName());
    };
};

class PersonPtr: public IElemPointer <Person> {

    friend inline Person& elementForOps (PersonPtr& A) {
        return (Person&) elementForOps ((IElemPointer<Person> &)A); }

    friend inline Person const& elementForOps (PersonPtr const& A) {
        return (Person const&) elementForOps ((IElemPointer<Person> &) A);}

public:
    PersonPtr(): IElemPointer<Person>() {}
    PersonPtr(Person* ptr,IExplicitInit IINIT)
        :IElemPointer<Person>(ptr,IINIT) {}
};

ostream& operator<<(ostream& os,Person A);

inline IString const& key (Person const& A)    //Key access
{ return A.GetPersonName();};
```

The following example shows the main file.

```
//main.cpp - main file
#include "person.h" //person.h from the previous example
#include <istdops.h>
#include <iset.h>

typedef IMngPointer <Person> ManagedPersonPtr;
typedef ISet <ManagedPersonPtr> AddressList;

ostream& operator<<(ostream& os,Person A) {
    return (os << endl << A.GetPersonName() <<" "<<A.GetTNumber());
}

void main() {
    AddressList Business;
    AddressList::Cursor myCursor1(Business);

    ManagedPersonPtr Aptr (new Person("Peter Black","714-50706"),IINIT);
    ManagedPersonPtr Bptr (new Person("Carl Render","714-540321"),IINIT);
    ManagedPersonPtr Cptr (new Person("Sandra Summers","x"),IINIT);
    ManagedPersonPtr Dptr (new Person("Mike Summers","x"),IINIT);
    ManagedPersonPtr CopyCptr (new Person("Sandra Summers","x"),IINIT);

    Business.add(Aptr);
    Business.add(Bptr);
```

```

Business.add(Cptr);
Business.add(Dptr);
Business.add(CopyCptr);

forICursor (myCursor1) {
    cout << *Business.elementAt(myCursor1);
}
}
/* Comment: CopyCptr and Cptr refer to different memory addresses, so
both of them are entered into the set even if the element they
point to is identical. This is because equality now refers to the
pointers even though it is also defined for Person. */

```

On the other hand, if you want element functions to work on the elements referenced by the pointers, the Collection Classes offer the `IElemPointer`, `IAutoElemPointer` and `IMngElemPointer` pointer classes, which are instantiated with the element type. Pointers of these classes automatically apply all element functions, except for assignment, to the referenced object. Element pointers are constructed from C++ pointers. The C++ dereferencing operators `*` and `->` are defined, for element pointers, to refer to the referenced objects. Consider the following example:

```

//main.cpp - main file
#include "person.h" //person.h from the previous examples
#include <istdops.h>
#include <iset.h>

typedef ISet <PersonPtr> AddressList;

ostream& operator<<(ostream& os, Person A) {
    return (os << endl << A.GetPersonName() << " "<< A.GetTNumber());
}

void main() {
    AddressList Business;
    AddressList::Cursor myCursor1(Business);

    PersonPtr Aptr (new Person("Peter Black", "714-50706"), IINIT);
    PersonPtr Bptr (new Person("Carl Render", "714-540321"), IINIT);
    PersonPtr Cptr (new Person("Sandra Summers", "x"), IINIT);
    PersonPtr Dptr (new Person("Mike Summers", "x"), IINIT);
    PersonPtr CopyCptr (new Person("Sandra Summers", "x"), IINIT);

    Business.add(Aptr);
    Business.add(Bptr);
    Business.add(Cptr);
    Business.add(Dptr);
    Business.add(CopyCptr);

    forICursor (myCursor1) {
        cout << *Business.elementAt(myCursor1);
    }

    Business.remove(Cptr); //Remove pointer from collection
    cout << "\nPointer was removed from collection but still exists : "
        << *Cptr;
    delete (Person*) Cptr;
}

/* Because PersonPtr is an ElementPointer, you must manually
free memory. */

```

The dynamically created elements are not automatically deleted when they are removed from the collection.

Managed Pointers

Managed pointers keep a reference count for each referenced object (element). When the last managed pointer to the object is destructed, the object is automatically deleted. You should use managed pointers when you are unsure who is responsible for deleting an object. This may occur where several pointers to an object are introduced over time, and the order in which the pointers are released is not known.

The following example shows how to use pointers from the `IMngElemPointer` class:

```
//main.cpp - main file
#include "person.h" //person.h from the previous examples
#include <istdops.h>
#include <iset.h>

typedef IMngElemPointer <PersonPtr> MEPersonPtr;
typedef ISet <MEPersonPtr> AddressList;

ostream& operator<<(ostream& os, Person A) {
    return (os << endl << A.GetPersonName() << " "<<A.GetTNumber());
}

void main() {
    AddressList Business;
    AddressList::Cursor myCursor1(Business);

    MEPersonPtr Aptr (new Person("Peter Black","714-50706"),IINIT);
    MEPersonPtr Bptr (new Person("Carl Render","714-540321"),IINIT);
    MEPersonPtr Cptr (new Person("Sandra Summers","x"),IINIT);
    MEPersonPtr Dptr (new Person("Mike Summers","x"),IINIT);
    MEPersonPtr CopyCptr (new Person("Sandra Summers","x"),IINIT);

    Business.add(Aptr);
    Business.add(Bptr);
    Business.add(Cptr);
    Business.add(Dptr);
    Business.add(CopyCptr);

    forICursor (myCursor1) {
        cout << *Business.elementAt(myCursor1);
    }

    Business.remove(Cptr); //Remove pointer from collection

    //delete (Person*) Cptr; //Wrong: after removing the pointer from the
                           //collection, the managed pointer is
                           //automatically deleted.
}
```

In the example, the allocated `Person` will automatically be deleted by the `remove()` function unless it is referenced through another `PersonPtr`.

Automatic Pointers

Automatic pointers do not keep a reference count. A referenced object (element) is automatically deleted in two cases:

- The automatic pointer is destructed. Automatic pointers should be used when the lifetime of the element is the same as the lifetime of the pointer, but when an explicit deletion of the element is awkward or even impossible. This applies in particular to pointers to objects that are dynamically created within a function, and whose lifetime is the scope of the function. The function may be left through several return statements or through an exception being thrown from some other function being called.

- Using the assignment operator, the automatic pointer is used to point to another element (which is implicitly a new element). The assigned pointer is set to NULL.

If you define a collection taking automatic pointers as elements, the elements are automatically deleted when the collection is destructed, when an element is removed, or, if the element was not added to the collection, when the variable or temporary holding the pointer is destructed. Consider the following example:

```
//main.cpp - main file
#include "person.h" //person.h from the previous examples
#include <istdops.h>
#include <iset.h>

typedef IAutoElemPointer <Person> AEPointer;
typedef ISet <AEPointer> AddressList;

ostream& operator<<(ostream& os, Person A) {
    return (os << endl << A.GetPersonName() << " "<<A.GetTNumber());
}

void main() {
    AddressList Business;
    AddressList::Cursor myCursor1(Business);

    Business.add(AEPointer (new Person("Peter Black","714-50706"),IINIT));
    Business.add(AEPointer (new Person("Carl Render","714-540321"),IINIT));
    Business.add(AEPointer (new Person("Sandra Summers","x"),IINIT));
    Business.add(AEPointer (new Person("Mike Summers","x"),IINIT));
    //The temporary automatic pointer variables were set to NULL
    //when the pointer was copied to the collection.

    {
        Business.add(AEPointer (new Person("Sandra Summers","x"),IINIT));
    } //Deletes the second Person ("Sandra ..."), because it was not
    //added (note that in a set, each element occurs only once).

    forICursor (myCursor1) {
        cout << *Business.elementAt(myCursor1);
    }
    //Deletes all pointers that were added previously to the set
    //with the destruction of the set.
}
```

Transfer of Automatic Pointers

You should be aware of the implementation details described below when transferring automatic pointers between functions. Consider the following cases:

- A calling function passes an automatic pointer to a called function and the pointer is returned.

```
IAutoPointer <Int> f (IAutoPointer <Int> i) { return i; }
// ...
main () {
    IAutoPointer <Int> i (new Int (5), IINIT);
    cout << *f(i) << endl;
}
```

This program results in the following taking place at runtime:

- **main** constructs an IAutoPointer object i and initializes it with the address of Int object 5.
- On invocation of f(), the copy constructor of IAutoPointer is called and the new constructed auto pointer is initialized with the address of the given input pointer. The given pointer is set to NULL. On return from f(), the copy constructor of IAutoPointer constructs a new auto pointer in main()

and initializes it with the address of the auto pointer object from `f()`, which then is destructed.

- When **main** exits, it calls the destructors for all auto pointer objects and the destructor for `Int` object 5.
- A called function has no input, but returns an object that has been dynamically created using an automatic pointer.

```
Int g() {  
    IAutoPointer <Int> j (new Int (6), IINIT);  
    return *j;  
}  
// ...  
main () {  
    cout << g() << endl;  
}
```

This program results in the following taking place at runtime:

- On invocation of `g()`, this function constructs an `IAutoPointer` object, constructs an `Int(6)` object, and initializes the auto pointer with the address of `Int(6)`.
- On return from `g()`, the copy constructor of `Int` constructs a new `Int(6)` object in `main()`. The auto pointer object and the `Int(6)` object in `g()` are destructed.
- On exit from `main()`, the `Int(6)` object is destructed.

Constructing Smart Pointers

All smart pointers have two constructors: a default constructor that initializes the pointer to `NULL`, and a constructor taking a C++ pointer to an element that you must have created before (using `new`).

Implicit conversions from a C++ pointer to a managed or automatic pointer are dangerous: elements might be implicitly deleted without your being aware that this has happened. Therefore, the conversion functions for these classes take an extra argument `IINIT` to make the construction explicit. Hence, the notation for creating a managed or automatic pointer is:

```
IAutoPointer < E > ePtr (new E, IINIT);
```

Note: After you have constructed a managed or automatic pointer from a C++ pointer, *you should no longer use the C++ pointer*. You should only access the element through the pointer of the given class. Otherwise, the element could be implicitly destructed while a C++ pointer still refers to it. In particular, you must not construct two managed pointers or two automatic pointers from the same C++ pointer, because this would cause the managed pointers to keep two separate reference counts, and to implicitly delete the referenced element twice. For example:

```
IStrng *s = new IStrng("...");  
IMngPointer < IStrng > p1 (s, IINIT); // OK  
IMngPointer < IStrng > p2 (s, IINIT); // NO!  
// Do not use s a second time, because the compiler may try to  
// delete the IStrng object referred to by s, p1, and p2 twice.
```

You should keep the following rule in mind when using managed or automatic pointers created from standard pointers: *Never use the C++ pointer once the managed or automatic pointer has been created from it*, because this may interfere with the automatic storage management. For example, the object that is referenced by a C++ pointer and by an automatic pointer created from this C++ pointer, is deleted as soon as the automatic

pointer gets out of scope. The C++ pointer then points to undefined storage.

The extra IINIT argument is introduced to make such situations explicit and especially to avoid the usage of the constructor as an implicit conversion operator. The IINIT argument is defined as follows:

```
enum IExplicitInit {IINIT};
```

Without the IINIT argument, you might try to write code such as the following:

```
//main.cpp - main file
#include "person.h" //person.h from the previous examples
#include <istdops.h>
#include <iset.h>

typedef IMngPointer <Person> MPointer;
typedef ISet <MPointer> AddressList;

ostream& operator<<(ostream& os, Person A) {
    return (os << endl << A.GetPersonName() << " "<<A.GetTNumber());
}

void func (MPointer aPointer);
//.....

void main() {
    AddressList Business;
    AddressList::Cursor myCursor1(Business);
    Person* ptr1=new Person("Peter Black","714-50706");
    MPointer mngdP=MPointer (ptr1);

    func (ptr1);    //Error: Second use of the C++ pointer.
}
// This listing is not intended to work. It illustrates how to
// avoid serious errors.
```

For the call to func(), the compiler would call a constructor for implicit conversion if the constructor did not require IINIT. On function return the temporary managed pointer would be destructed and the Task object deleted.

Notes on Smart Pointers

1. The smart pointers do not work with basic types such as **int**, **long**, and **char**.
2. If you implement a key collection containing element pointers, you must define your key() function with the element as input, not the pointer to the element, for example,

```
typedef IKeySortedSet <IMngElemPointer <Element>, int> keySortedSetOfPointers;
// ...
int const& key(Element const& element) {
    return element.elementKey();
}
```

where elementKey() returns the element's key.

3. An automatic pointer's copy constructor and assignment operator are defined in a way that resets the source pointer to NULL. This prevents multiple automatic pointers from pointing to the same element. In the following example, p2 is implicitly set to NULL:

```
IAutoPointer < E > p1, p2;
...
p1 = p2;
```

However, the copy constructor and assignment operator still take a const argument (using a const cast-away) to maintain compliance with the standard

interface for these operations. This standard interface is required, for example, when you use these types as element types in collections, because the copy constructor and assignment operator are required to have such an interface. (Otherwise, the collection's `add()` function could not take a `const` argument.)

4. If you want to create managed pointers for a collection and copy in elements from a second collection that already contains managed pointers, you cannot use `IINIT` because it will destroy the managed pointers in the second collection. To avoid this situation, you can use the following notation:

```
typedef IMngElemPointer <PersonPtr> MyClassPtr;  
typedef IKeySet<MyClassPtr> MyAddressList;
```

```
MyClassPtr pMyClass;  
:pMyClass = Business.elementWithKey(...);
```

In the above notation, `Business` is the collection from the previous examples, but here it is an `IKeySet` collection rather than an `ISet` collection so that `.elementWithKey` can be used.

Chapter 10. Tailoring a Collection Implementation

This chapter describes how to tailor a collection implementation for your specific applications. It describes the based-on concept and predefined implementation variants.

Introduction

When you are developing a program that uses a collection, you should begin by using the default implementation and go on to a final tuning phase where you choose implementations according to the actual requirements of your application. You can determine these requirements by profiling or by using other measurement tools. This section describes how to choose between a variety of implementations provided by the Collection Classes as well as how to create your own implementation classes.

As described in “The Overall Implementation Structure” on page 83, each abstract data type has several possible implementations. Some of these implementations are *basic*; that is, the collection class is implemented directly as a concrete class. These basic implementations include:

- AVL trees
- Hash tables
- Linked sequences
- Tabular sequences

Other implementations, including bags, are *based on* other collection classes. The based-on concept provides a systematic framework for choosing the most appropriate implementations. It is also useful for extending the Collection Classes with other basic implementations, such as specific kinds of search trees, and for using these implementations as the basis for other data abstractions such as sets, maps, and bags.

Replacing the Default Implementation

You can easily replace the default implementation with another implementation. Suppose that you have a key set class called `MyType` that has been defined with the default implementation `IKeySet`. The definition of this class would look like this:

```
typedef IKeySet < Element, Key > MyType;
```

If you want to replace the default implementation, which uses an AVL tree, with a hash table implementation, you can replace the above implementation with the following definition:

```
typedef IHashKeySet < Element, Key > MyType;
```

If you replace a collection's default implementation with one of its implementation variants, you must determine what element functions and key-type functions need to be provided for the variant. You must then provide those functions. The list of required functions is not always the same for a collection's default implementation as for particular implementation variants. Required functions for a collection's default implementation or an implementation variant are listed in the collection's

chapter in the *OS/390 C/C++ IBM Open Class Library Reference*. See the section “Template Arguments and Required Functions” in each such chapter.

The Based-On Concept

The Collection Classes achieve a high degree of implementation flexibility by basing several collection class implementations on other abstract classes, rather than by implementing them directly through a concrete implementation variant of the class. This design feature results in an implementation path rather than the selection of an implementation in a single step. The Collection Classes contain type definitions for the most common implementation paths; they are described in the corresponding sections of the *OS/390 C/C++ IBM Open Class Library Reference*. See Figure 13 on page 123 for an illustration of implementation paths. The figure is explained in “Provided Implementation Variants.”

The element functions that are needed by a particular implementation depend on all collection class templates that participate in the implementation. While `ISet` requires at least element equality to be defined, an AVL tree implementation of this set also requires the element type to provide a comparison function. A hash table implementation also requires the element type to have a hash function. The required element functions for all predefined implementation variants are listed in the chapters for individual collection types in the *OS/390 C/C++ IBM Open Class Library Reference*.

For a concrete implementation, such as a set based on a key-sorted set that is in turn based on a tabular sequence, these class templates are *plugged* together.

Provided Implementation Variants

Figure 13 on page 123 lists the basic and based-on implementations provided by the Collection Classes. The upper left corner of each cell contains the name of the (abstract) collection class; basic implementations are written in smaller letters in bold face, while based-on implementations are described by arrows starting from the class that they implement and ending in the (abstract) class on which they are based. An implementation choice for a given class must use either a basic implementation for this class or follow a based-on implementation path that ultimately leads to a basic implementation.

Take the example of the `Set` abstraction. The `Set` is not implemented directly. (You can tell this from the figure because no implementation variant name appears in bold in the box containing `Set`.) To determine the possible implementation variants for `Set`, follow the arrows out of the `Set` box:

- One arrow leads to the `KeySet` box. The `KeySet` box contains an implementation variant, **Hash Table**, so this is one possibility. An arrow also points from the `KeySet` Box to the `KeySortedSet` box, which allows the following possibilities:
 - **AVL Tree** (appears in `KeySortedSet` box)
 - **B* Tree** (appears in `KeySortedSet` box)
 - An arrow leads from `KeySortedSet` to `Sequence`, which contains the following implementation variants:
 - **List**
 - **Table**
 - **Diluted Table**

A Set can therefore be implemented using any of the six implementation variants cited in bold face above.

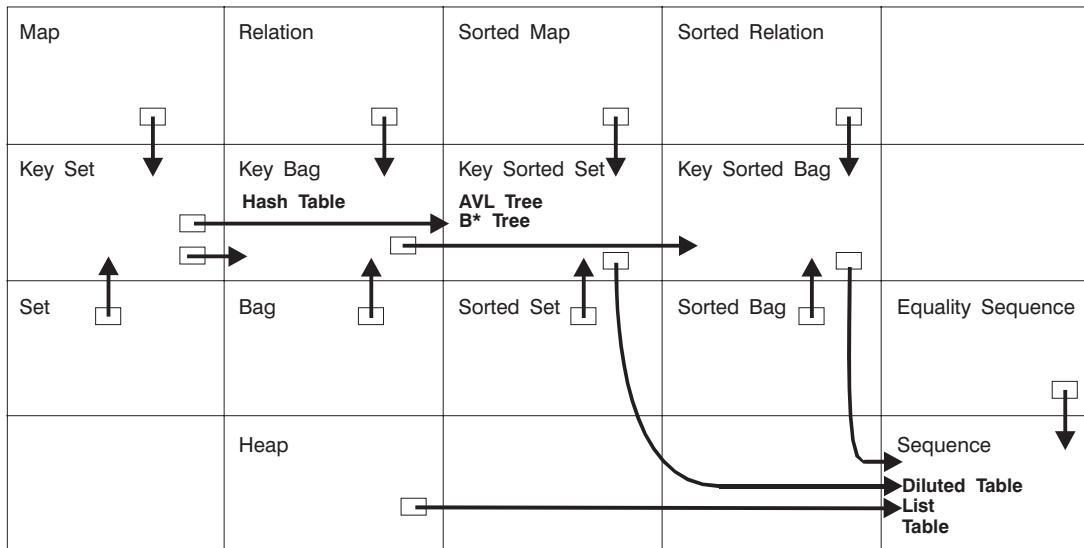


Figure 13. Possible Implementation Paths

Table 6. Implementation Variants

Implementation Variant	Bag	Sorted Bag	Key Bag	Key Sorted Bag	Set	Sorted Set	Key Set	Key Sorted Set	Map	Sorted Map	Relation	Sorted Relation	Sequence	Equality Sequence	Heap
AVL Tree					■	■	■	■	■	■					
B* Tree					●	●	●	●	●	●					
Hash Table	●		■		●		●		●		■				
List	■	■	●	■	●	●	●	●	●	●	●	■	■	■	■
Table	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
Diluted Table	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●

Figure 14. Implementation Variants Provided for Each Flat Collection. Squares identify default implementations; circles identify implementation variants.

Features of Provided Implementation Variants

You can implement a given collection type (bag, key sorted set, etc.) in a number of different ways. The possible implementation variants are described in “Provided Implementation Variants” on page 122, and are listed in the “Class Implementation Variants” section of each collection chapter in the *OS/390 C/C++ IBM Open Class Library Reference*. The Collection Classes provide multiple implementation variants for collections because different variants have different performance and storage use characteristics. After you have coded and debugged an application that uses

the Collection Classes, you can change an implementation to a variant that is well-suited to the ways in which you use the collection. For example, in Chapter 21, “Key Set” in the *OS/390 C/C++ IBM Open Class Library Reference*, the section “Variants and Header Files” on page 151 lists six implementation variants, including the default key set. These variants are implemented using the following concrete techniques:

- Sequences
 - List
 - Table
 - Diluted table
- Trees
 - AVL tree (the technique used for the default key set)
 - B* tree
- Hash table

As it turns out, the implementation variants for key set encompass all the concrete techniques used by the Collection Classes. Other collections may only use some of the techniques in the list above. If you want to choose the best implementation variant for your program, you need to know the advantages of each concrete technique. The remainder of this section describes each technique and presents its advantages and the trade-offs it entails.

Sequences

Sequences are generally used to store elements sequentially. Each of the three available implementation variants for sequences allows certain operations to be done more efficiently than others. The benefits of each variant are described first, and then each variant is explained in detail.

Lists are suitable when you anticipate that many elements will be added or deleted, and where you cannot accurately predict the maximum size of the collection when it is first created.

Tables provide good performance where a collection is primarily used for reading data but elements are not frequently added or deleted once the collection is created.

Diluted tables are more suitable for collections where some elements are inserted or deleted after the collection is created, but where the collection is still primarily read from rather than written to.

Following are descriptions of each type of sequence.

List

A *list* uses pointers to link each element to its predecessor and successor. This implementation does not require contiguous memory for storing an array, which means that elements do not have to be shifted to make room for new elements or to close up gaps created by deleted elements.

Because storage is dynamically allocated and freed, this implementation variant is a good choice in applications that add or delete many elements, particularly where you cannot predict the amount of storage required. Figure 15 on page 125 shows a list implementation variant.

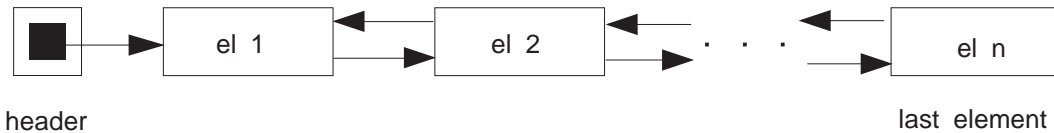


Figure 15. List Implementation Variant

Table

A *table* is an array implementation of a sequence. The elements are stored in contiguous cells of an array. In this representation, a list can easily be traversed, and new elements can easily be added to the tail of the list. If an element needs to be inserted into the middle of the list, however, all following elements need to be shifted to make room for the new element. Similarly, if an element needs to be removed from the list, and the element is not the last element in the list, all elements following the element to be deleted must be shifted in to close up the gap.

A table can access all elements quickly because all elements can be stored in a single storage block. If all of the following conditions hold true for your use of a collection, a table is a suitable implementation variant to use:

- The elements to be stored are small.
- You can predict with some accuracy how many elements your application will have to handle.
- Few or no elements will need to be added or deleted once the collection is first created.

Note that memory is statically allocated for tables, at the beginning of your program.

Figure 16 shows a table implementation variant.

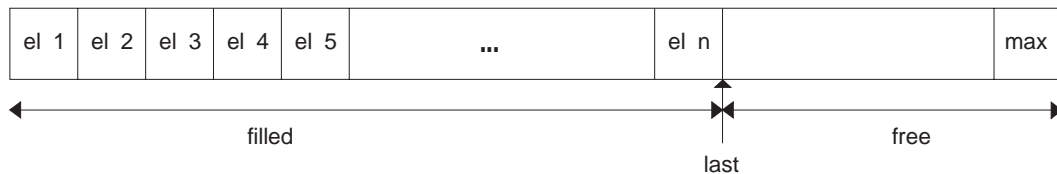


Figure 16. Table Implementation Variant

Diluted Table

A *diluted table*, like a table sequence, is an array implementation of a list. However, when you delete an element from a diluted table, it is not actually deleted, but only flagged as deleted. This provides a performance advantage, in that elements following a deleted element do not need to be shifted. The additional overhead of using a dilution flag is trivial.

If you want to add a new element at a certain position, only those elements between that position and the next element flagged as deleted need to be shifted. (If no elements later in the list are flagged as deleted, then all elements beyond the insertion position must be shifted.)

Provided Implementation Variants

Use a diluted table rather than a table if your application will be doing much adding or deleting of elements after the collection is established.

Figure 17 shows a diluted table implementation variant.

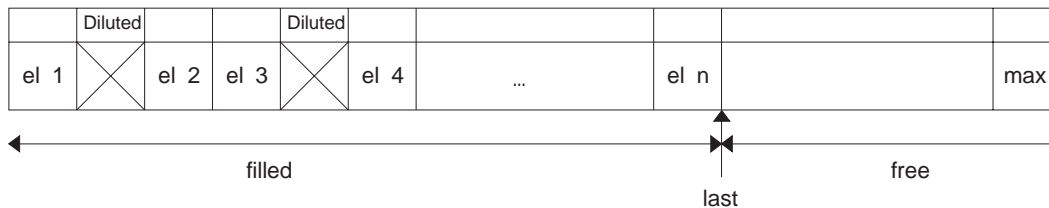


Figure 17. Diluted Table Implementation Variant

Trees

A tree is a collection of nodes. The nodes either contain the data of the collection or pointers to that data.

A node normally contains a reference to one or more other nodes. Referenced nodes are *children* of the referencing node. One node is the entry point to the tree. This node is designated as the *root*. Nodes without any references to other nodes are called *leaf nodes* or *terminal nodes*.

Trees in general are more useful for searching elements than for adding and deleting elements. For this reason, they are often called *search trees*. The descriptions of AVL and B* trees below explain why trees are well-suited for searching.

AVL Tree

AVL trees are a special form of binary tree. You can better understand AVL trees if you know how a binary tree is structured.

Trees are *binary trees* when all nodes have either zero, one, or two children. Binary trees are often used in applications where you want to store elements in a certain order. In such cases, the left child always points to an element that comes earlier in the order than the parent node, and the right child points to an element that comes later than the parent. A search through a binary tree begins at the root node. The search then continues downward until the desired element is found, by determining whether a node comes before or after the searched-for node, and then following the appropriate branch. For example, the binary tree shown in Figure 18 on page 127 has elements added in the following sequence: 8 - 10 - 5 - 1 - 9 - 6 - 11. A search for element 9 begins at the root node (element 8). Assuming that the element value defines the ordering relation, the search would take the right node from element 8 (because 9 is greater than 8) and would arrive at element 10. The search would take the left node from element 10 (because 9 is smaller than 10) and would arrive at element 9, the desired element.

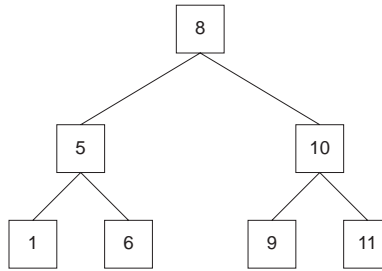


Figure 18. Binary Search Tree

One drawback of a binary search tree is that the tree can easily become unbalanced. Figure 19 shows how unbalanced the tree becomes when the elements 12 through 15 are added.

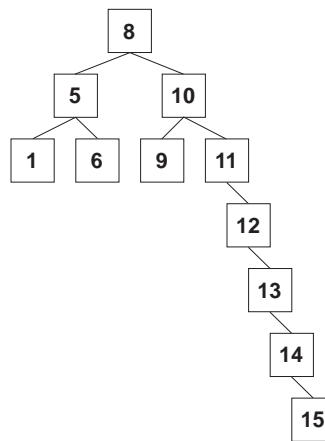


Figure 19. Unbalanced Binary Search Tree

This tree looks almost like a list, without the performance advantage of a normal binary search tree. To obtain this performance advantage, a binary search tree should always remain balanced. The *AVL Tree* is a special form of binary search tree that maintains balance.

The *AVL tree* was invented by the two mathematicians, Adel'son-Vel'skii and Landis, from whom it derives its name. AVL trees are *height-balanced*. They have the property that, for every node in the tree, the height of that node's left subtree minus the height of the right subtree is always -1, 0, or +1. AVL trees provide better performance than ordinary binary search trees because they do not become unbalanced. Unbalanced trees often have very poor search characteristics. If adding or removing an element from an AVL tree causes the tree to lose its AVL property, then a few local readjustments are sufficient to restore the AVL property. Figure 20 on page 128 shows how the unbalanced tree shown earlier would look after the AVL property is restored.

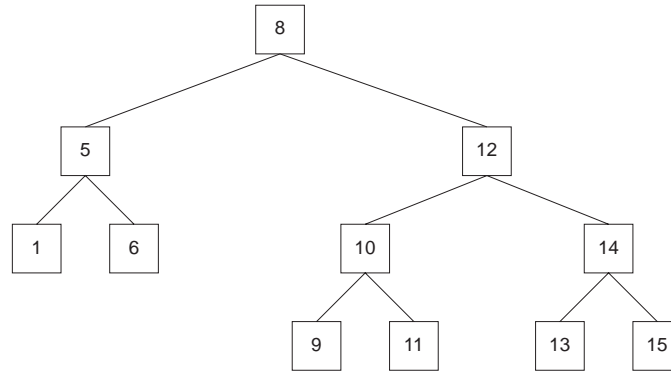


Figure 20. AVL Tree

AVL trees are useful for collections containing a large number of small elements. An AVL tree implementation is even suitable for adding and deleting, because the performance overhead for the rebalancing that sometimes occurs when an element is added or deleted is still less expensive than searching through the elements of a sequence to find the position at which to add or delete an element.

If you use a set collection and do not choose an implementation variant, you are automatically using an AVL tree. If you use a set and are not aware that the set is implemented as an AVL tree, you may be surprised that a set requires an ordering relation, when a set is an *unordered* collection, as shown in Figure 8 on page 78. The reason a set requires an ordering relation is that an AVL tree requires an ordering relation so that it knows where to add new elements or where to find elements being accessed or deleted. As this example shows, required element and key-type functions are determined by two factors:

- Some functions are required because of the properties of the collection.
- Some properties are required because of the implementation variant you choose.

B* Tree

A *B* tree* is a search tree that may have more than two references per node. Figure 21 shows a B* tree with up to five children per node.

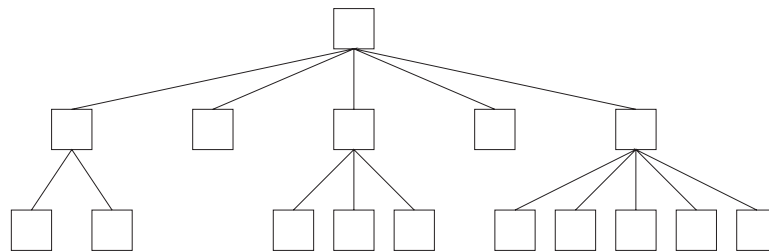


Figure 21. A B* tree

A B* tree combines the advantages of binary search and sequential access upon the same set of keys. B* trees are based on two simple ideas:

- The internal nodes are used only for storing the keys, with all real data stored at the leaves. A B* tree takes into consideration the page or block size of the operating system's virtual memory structure, and is suitable for applications where paging or memory thrashing is a constraint.

- The leaves of a B* tree are chained together in logical sequence to support sequential access.

A B* tree implementation variant is suitable when you have many large elements that are accessed by key. Because keys and their data are separated, the keys in the tree structure are used for a quick search and the pointers are used for quick access to the data.

In contrast to a B* tree, keys and data in an AVL tree are both stored in the nodes. This means that searching through elements could cause page faults if the elements are large, because the various keys may be spread across several pages along with the data they refer to.

In Figure 22, the B* tree has an order of 5 (which means that each internal node has a maximum of five references). The data is stored only in the leaves. A leaf block is built to hold one element. A leaf block may be larger than one page. The B* tree implementation uses the keys in the nodes for quick access to a required page (leaf), or it uses the keys for a quick sequential access to all pages, and hence to all elements.

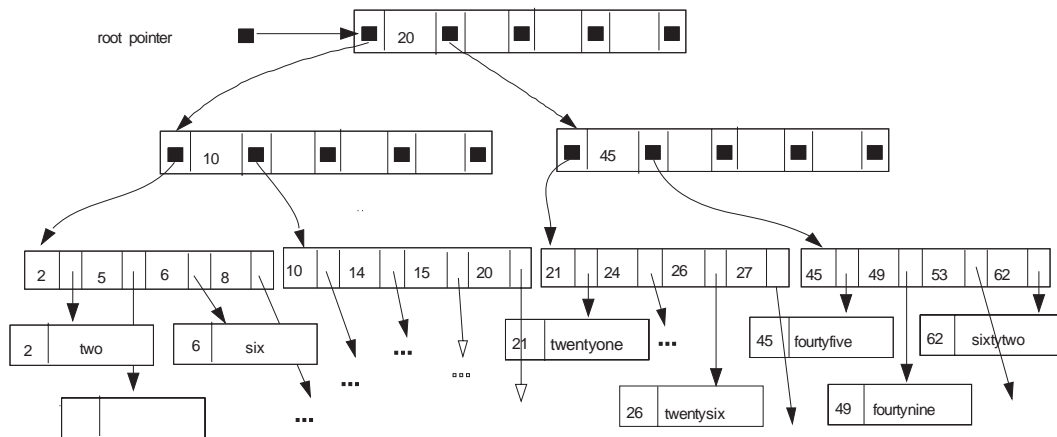


Figure 22. B* Tree Implementation Variant

Hash Table

Hashing is another important and widely used technique to implement collections. Conceptually, hashing involves calculating an index from the key or other parts of an element, and then using that index to look for matches in a hash table. The function that calculates the index is called a *hash function*.

A hash table implementation variant is suitable for nearly all applications with a balanced mix of operations. Such an implementation is quick for retrieving elements. It can also add and delete elements quickly, because, unlike an AVL tree, it does not need to be rebalanced. The efficiency of a hash-table implementation is largely dependent on how efficiently you implement the hash function.

You cannot use a hash-table implementation variant when you require your elements to appear in main storage in sorted order (where elements earlier in the

Provided Implementation Variants

sorting order have lower addresses than elements later in the sorting order). On the other hand, you must use a hash table if you have a complex key (one composed, for example, of several attributes of an element), and either you cannot find a reasonable way to compare keys, or the comparison would be expensive.

For collections that do not provide access by key, but that support a hash-table implementation variant, the complete element is used as the input to the hash function.

Hashing, as implemented in the collection classes, allows elements to be stored in a potentially unlimited space, and therefore imposes no limit on the size of the collection. Figure 23 shows a hash table implementation variant.

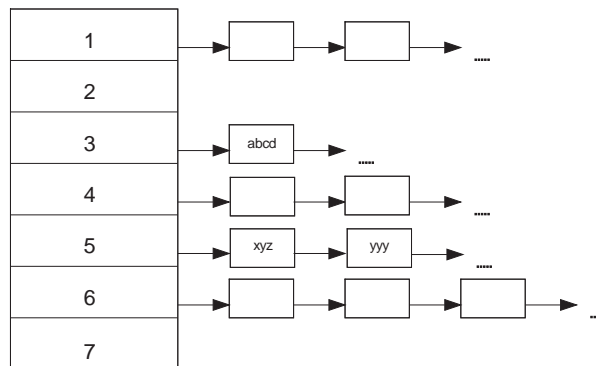


Figure 23. Hash Table Implementation Variant

The hash function that calculates the index 3 from *abcd* is implemented as follows:

1. Each character is transformed into an integer according to its position in the alphabet.
2. The resulting integers are added together.
3. The result is divided by the hash table size. The remainder is the hash.

This hash function returns the following results for elements *abcd*, *xyz* and *yyy*:

- *abcd*: $(1 + 2 + 3 + 4) \% 7 = 3$
- *xyz*: $(24 + 25 + 26) \% 7 = 5$
- *yyy*: $(25 + 25 + 25) \% 7 = 5$

The principal behind a hash table is that the possibly infinite set of elements in your collection is partitioned into a finite number of hash values (1, 2, 3, ...). Your hash function is called with a key and a modulo value, and you use the key and the modulo value to arrive at an integer hash value. If for two different keys the hash function returns the same hash value (as for *xyz* and *yyy* in the previous figure), a hash *collision* occurs. In such cases, a hash implementation constructs a collision list where all keys returning the same hash value are linked.

In the best case, for each different key, your hash function should return a different hash value. At the very least, it is desirable for the collision lists to remain small so that access time is fast. This means that hash values should be evenly distributed. Your hash function should randomly hash the key so that the hash value is not dependent on the key value in any trivial way. Your hash function should always return the same hash value for a given key and modulo provided to it.

Chapter 11. Polymorphism and the Collections

This chapter describes how you can make use of polymorphism in the Collection Classes

Introduction to Polymorphism

Polymorphism allows you to take an abstract view of an object or function argument and use any concrete objects or arguments that are derived from this abstract view. The collection properties defined in “Flat Collections” on page 78 define such abstract views. They are represented in the form of the class hierarchy in Figure 11 on page 86.

Polymorphic use of collections differs from polymorphism of the element type. Polymorphic use of collections means that a function can specify an abstract collection type for its argument, for example `IACollection`, and then accept any concrete collection given as its actual argument. Element polymorphism means that you can use the collections with any elements that provide basic operations like assignment and equality. This chapter deals with the polymorphic use of collections rather than elements.

Each abstract class is defined by its functions and their behavior. The most abstract view of a collection is a container without any ordering or any specific element or key properties. Elements can be added to a collection, and a collection can be iterated over. A polymorphic function on collections that uses only properties of the most abstract view might be to print all elements; such a function is given as an example on page 132.

Collections with more specialized element properties, such as equality or key equality, also provide functions for retrieving element occurrences by a given element or key value. Ordered collections provide the notion of a well-defined ordering of element occurrences, either by an element ordering relation or by explicit positioning of elements within a sequence. Ordered collections define operations for positional element access. Sorted collections provide no further functions, but define a more specific behavior, namely that the elements or their keys are sorted.

The properties represented by abstract collection classes are combined through multiple inheritance: The abstract collection class `IAEqualitySortedCollection`, for example, combines the properties of element equality and of being sorted, which implies being ordered. If a polymorphic function uses `IAEqualitySortedCollection` as its argument type, the argument will be sorted, and the function can use functions such as `contains()` that are only defined for collections with element equality.

Using the Abstract Class Hierarchy

The following example defines a universal printer class that accepts an arbitrary collection of jobs and prints their IDs. The elements are printed in the iteration order that is defined for the given collection. The key set `running` can be used as argument to the universal printer.

```
class JobPrinter {
public:
    print (ICollection <Job*> const& jobs)
    { cout << "ID      ..."
      ICursor *cursor = jobs.newCursor ();
      cout << "{ ";
      forICursor (*cursor)
          cout << jobs.elementAt (*cursor)->id() << ' ';
      cout << "}\n";
      delete cursor;
    }
};
// ...
typedef IKeySet <Job*, JobId> JobSet;
JobSet running;
// ...
JobPrinter jobPrinter;
jobPrinter.print (running);
```

Adding and Overloading Member Functions

When you derive classes from the Collection Classes you can do so in two different ways:

1. The derived class only adds new member functions.
2. The derived class overloads existing member functions. The derived collection class will not be used in a polymorphic way.

You do not need to take any special measures. You just code your derived class as usual. For example, suppose you want to implement a set of integers that can give you information about the sum of integers contained in the collection. You create a class `IntSet` that is derived from `ISet<int>`. This class does the following:

1. Introduces the data member `ivSum` to hold the current sum.
2. Adds the member function `sum()`, which returns the current sum.
3. Overloads the `add()` member function so that it updates `ivSum` each time an integer is added to the collection.

In a real application, any `add`, `replace` or `remove` member function would have to be overloaded in order to update the sum of integers. For simplicity, this is not done in the example below:

```
#include <iset.h>

class IntSet: public ISet<int> {
    typedef ISet<int> Inherited;
public:
    IntSet (INumber n = 100)
        : ISet<int> (n), ivSum (0) { }
    IBoolean add (int const& i)
        { ivSum += i;
          return Inherited::add(i); }
    int sum () const
        { return ivSum; }
```

```
private:
    int      ivSum;
};

// ...
IntSet anIntSet;
anIntSet.add (1);
anIntSet.add (2);
cout << anIntSet.sum () << endl;
```

The output of this program is 3.

Note: Collection classes do not have virtual functions. You cannot override the member functions of a collection class.

Chapter 12. Support for Notifications

The Collection Classes include special classes that support notifications. For every concrete flat collection class (for example `ISequence`), there is a corresponding notification-enabled collection class starting with `IV` (for example `IVSequence`).

All collection methods that modify a collection send notifications to observers. The class `IVCollection` defines four notification IDs for Collection Classes:

<code>IVCollection: addId</code>	Sent if an element is added to the collection
<code>IVCollection: removeId</code>	Sent if an element is removed from the collection
<code>IVCollection: replaceId</code>	Sent if an element is replaced in the collection
<code>IVCollection: modifyId</code>	Sent if a collection is changed in any way other than those mentioned above.

The Collection notifications `addId`, `removeId` and `replaceId` pass a pointer to an instance of the class `IVCollectionEventData`.

The class `IVCollectionEventData` provides the methods:

1. `ICursor const& cursor() const`
2. `Element const*const element() const`

For the notifications `addId`, `removeId` and `replaceId`, you can use `INotificationEvent::eventData()` to access event data (`IVCollectionEventData`) generated by collections. The cursor points to the element referred to by the modification method. For example, if `addId` is the notification, the cursor points to the added element. The `replaceId` notification also gives you access to a copy of the element that was replaced.

For the notifications `addId` and `modifyId`, the library sends notification after the modification occurs. For the notification `removedId` and `replaceId` the library sends notification before the collection is changed, otherwise you would not be able to use the cursor to refer to the element being removed.

Notifications are only sent if the collection is changed by the method. The following methods do not create a notification:

- `removeAll()` for an empty collection
- `add()`, when `add()` does not actually add an element (for example, because the element already exists in a unique collection, or because the collection is full)
- `remove()` if the element is not in the collection
- `locateOrAdd()` if the element is already in the collection

Information about the notification framework is found in Chapter 23, "The IBM Open Class Notification Framework" on page 235.

Example for IVSequence<IString>

The following example demonstrates the use of collection event data for a sequence of IStrings. IString is the main string handling class provided by the IBM Open Class Library. See Chapter 19, “String Classes” on page 199 for information on how to use this class.

```
#include <iobserverv.hpp>
#include <inotifev.hpp>
#include <iseq.h>

typedef IVSequence<long> Notifier;

#include <iostream.h>

template <class Notifier>
class Observer : public IObserver {
public:

    Observer (Notifier* notifier)
    : ivNotifier (notifier)
    { handleNotificationsFor (*ivNotifier);
    }

    ~Observer ()
    { if (ivNotifier != 0) // critical !
      stopHandlingNotificationsFor (*ivNotifier);
    }

    IObserver&
    dispatchNotificationEvent (INotificationEvent const& event)
    { if (event.notificationId () == IStandardNotifier::deleteId) {
      cout << "IStandardNotifier::deleteId received" << endl;
    }
    else
    if (event.notificationId () == IVCollection::removeId) {
      cout << "IVCollection::removeId received" << endl;
      cout << "Old Data: "
          << ((IASequence<long>)(event.notifier()))
//
// IASequence can be either replaced by IACollection
// or IAOrderedCollection:
//          << ((IACollection<long>)(event.notifier()))
//
          .elementAt(((IVCollectionEventData<long>*)
              ((char*)event.eventData()))->cursor())
          << endl;
    }
    else
    if (event.notificationId () == IVCollection::replaceId) {
      cout << "IVCollection::replaceId received" << endl;
      cout << "Replace at position: "
          << ((IASequence<long>)(event.notifier()))
//
// IASequence can be replaced by IAOrderedCollection
//          << ((IAOrderedCollection<long>)(event.notifier()))
//
          .positionAt(((IVCollectionEventData<long>*)
              ((char*)event.eventData()))->cursor())
          << endl;
      cout << "Old Data: "
          << ((IASequence<long>)(event.notifier()))
//
// IASequence can be either replaced by IACollection
```

```

// or IAOOrderedCollection:
//     << ((ICollection<long>)(event.notifier()))
//
//         .elementAt(((IVCollectionEventData<long>*)
//         ((char*)event.eventData()))->cursor())
//     << endl;
//     cout << "New Data: "
//     << *(((IVCollectionEventData<long>*)
//     ((char*)event.eventData()))->element())
//     << endl;
// }
// else
// if (event.notificationId () == IVCollection::addId) {
//     cout << "IVCollection::addId received" << endl;
//     cout << "Add at position: "
//     << ((IAOrderedCollection<long>)(event.notifier()))
//     .positionAt(((IVCollectionEventData<long>*)
//     ((char*)event.eventData()))->cursor())
//     << endl;
//     cout << "New Data: "
//     << ((IASequence<long>)(event.notifier()))
// }
// IASequence can be either replaced by ICollection
// or IAOOrderedCollection:
//     << ((ICollection<long>)(event.notifier()))
// }
//         .elementAt(((IVCollectionEventData<long>*)
//         ((char*)event.eventData()))->cursor())
//     << endl;
// }
// else {
//     cout << "unknown event received" << endl;
// }
// return *this;
// }

private:

    Notifier* ivNotifier;

};

int main ()
{
    Notifier* n = new Notifier;
    Notifier::Cursor c(*n);
    Observer <Notifier> o (n);

    n->enableNotification ();
    {
        n->add(123,c);
        cout << "element in collection: " << n->elementAt(c) << endl;
        n->replaceAt(c,456);
        cout << "element in collection: " << n->elementAt(c) << endl;
        n->removeAt(c);
        cout << "Number of elements in collection: " << n->numberOfElements() << endl;
    }
    delete n;

    return 0;
}

```


Chapter 13. Thread Safety and the Collection Classes

Like most of the IOC classes, the collection classes require thread safe operation of multithreaded access to global data. The collection classes offer support for Level 1 thread safety, but they also offer built-in support to simplify the explicit serialization needed by the programmer to protect the collection instance. While serialization for global data is still needed under Level 1, the built-in support helps to reduce the amount of programming required. Note, however, that the locking of elements stored within a collection is the responsibility of the user and is not provided as part of collection class thread safety.

Guard Objects

For each different collection abstraction, a Guard class similar to `IResourceLock` has been defined and a corresponding typedef added:

```
template <class Element> class ICollectionGuard {}
typedef ICollectionGuard<Element> Guard;
```

Essentially, a Guard object is an object created on a stack that is used to lock some other object. Guard objects are useful in C++ because they respond properly to exceptions. When an exception is thrown while still in the scope of the Guard object, its destructor is called as the exception passes through the stack frame and the destructor unlocks the target object. As a result, the exception can be caught and dealt with by code further up the call chain without leaving the locked object in an unusable locked state.

The Guard typedef can be used as if it was a nested class of a particular collection and is based on one of three new classes added to the IBM collection class wrapper:

```
template <class Element>
class ICollectionGuard
{
public:
    ICollectionGuard(
        IACollection<Element>&, long timeout=-1);
    ~ICollectionGuard();
private:
    IACollection<Element>& ivCollection;
}
```

Note: The time-out parameter is ignored on OS/390.

Usage

In a user program, a Guard is used in the following way to obtain a lock on a specific collection:

```
//...
{ ISet<char> my_set;
  try {
    ISet <char>::Guard g(my_set);
    my_set.add('x');
  } catch (IException& e) {
    // The user's error recovery...
  }
}
//...
```

The critical region, in this case the add method invoked on the ISet<char>, must be specified within a C++ compound statement. On entry to the block, the Guard constructor locks the collection that is specified as the Guard constructor parameter. The destructor is executed when the scope of the block is left at the time the collection is unlocked. The specified name of the Guard object (g in the above example) is arbitrary and plays no role in the locking.

Depending on the number of threads of a particular user application, multiple Guard objects may exist that work with the same collection object.

For the Restricted Access Collections and the Tree Collections, two similar Guard classes and corresponding typedefs are added. They are exposed to the user through the following typedefs on the level of appropriate concrete collections:

```
typedef IRestrictedAccessCollectionGuard<Element> Guard;
typedef ITreeGuard<Element> Guard;
```

In the event that the user invokes a Collection method that involves two or even three collections, code such as the following must be used in order to achieve thread-safe execution:

```
//...
{ try {
  ISet <char>::Guard l1(my_set1);
  ISet <char>::Guard l2(my_set2);
  my_set1.addAllFrom(my_set2);
} catch (IException& e) {
  // The user's error recovery...
}
}
//...
```

In the case of three involved collections, the following code must be used:

```
//...
{ try {
  ISet <char>::Guard l1(my_set1);
  ISet <char>::Guard l2(my_set2);
  ISet <char>::Guard l3(my_set3);
  my_set1.addInterSection(my_set2,my_set3);
} catch (IException& e) {
  // The user's error recovery...
}
}
//...
```

The programmer does not need to include any new header files. The typedef for the ISet coding samples illustrated above is provided by the standard include file `iset.h`.

ICollectionGuard<Element> Constructor and Destructor

The Guard 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 to indicate an indefinite wait. The value 0 informs the constructor to throw an exception if the lock is not immediately available.

This parameter is only supported on non-POSIX platforms. Other platform implementations ignore the specification of this value. It is specifically ignored on OS/390.

The Guard destructor unlocks the Collection specified within the constructor of the Guard.

Guard Copy Constructor

The Guard copy constructor is made private in order to prevent the user from copying Guard objects.

Collection Constructor and Destructor

The collection does not keep track of all possible Guard objects currently in use with the target collection. Guards for a collection must be destructed before the collection itself is destructed. This is normally accomplished by declaring the Guard within a compound statement so that it is automatically destructed when the statement passes out of scope.

Collection Copy Constructor

If a new collection is created from an existing collection instance, the guards of the existing collection have no effect on the new collection.

Return Codes and Exceptions

Since the Guard is constructed, there are no return codes. The Collection classes use exceptions to indicate that a lock cannot be obtained. The user must code the Guard constructor within a try/catch clause. When the Guard constructor fails and the lock was not obtained for any reason, a C++ exception is thrown.

Deadlocks

In either of the above cases, you are responsible for the proper sequence of obtaining the locks. There is no special code within the collection classes to prevent the user from producing deadlocks.

Restrictions

The current implementation does not provide any means to support users who want to program in a multiprocessing environment with the IBM Open Class collection classes having collection instances in shared memory regions executed. It only supports thread safety within a single process.

Restrictions

Due to inherent POSIX limitations, the IBM Open Class collection classes do not support the described time-out processing in the following environments:

- OS/390 UNIX System Services
- AIX

The time-out parameters on OS/390 are ignored.

All guard processes that run on a non-OS/390 UNIX system are actually no operation (NOOP). The internally used IResource classes are not available on non-OS/390 UNIX systems.

Chapter 14. Exception Handling

This chapter describes the exception-handling facilities provided by member functions of the Collection Class Library. This chapter includes the following topics:

- Introduction to exception handling
- Preconditions and defined behavior
- Levels of exception checking
- List of exceptions
- The hierarchy of exceptions

Introduction to Exception Handling

The C++ exception-handling facilities allow a program to recover from an *exception*. An exception is a user, logic, or system error that is detected by a function that does not itself deal with the error, but passes the error to a handling function. Exceptions can result from two major sources:

- The violation of a precondition
- The occurrence of an internal system failure or system restriction

In this chapter, two kinds of functions are discussed. A *called* function is a Collection Class function that may throw an exception. A *calling* function is a function that calls a Collection Class function. The *calling* function may be a Collection Class function or a function you have defined.

Exceptions Caused by Violated Preconditions

A *precondition* of a called function is a condition that the function requires to be true when it is called. The calling function must assure that this condition holds. The called function implementation may assume that the condition holds without further checking it. If a precondition does not hold, the called function's behavior is undefined.

If you want to make your programs more robust and to locate errors in the test phase, the functions your program calls should check to ensure that their preconditions hold. The Collection Class Library enables this checking through macro definitions. Because this checking often requires significant overhead, it is turned off by default. You need only use it while you are testing the system and verifying that preconditions are always met.

A call to a function that violates the function's preconditions has two possible results:

- If the called function checks its preconditions, the function will throw an exception.
- If the function does not check its preconditions, the behavior of the function is undefined.

Exceptions Caused by System Failures and Restrictions

System failures and restrictions are different from precondition violations. You cannot usually anticipate them, and you have no opportunity to verify that such situations, for example storage overflow, will not occur. These exceptions need to be checked for, and an exception should be thrown if they occur.

Precondition and Defined Behavior

Exceptions are not generally used to change the flow of control of a program under normal circumstances. An example of using exceptions under normal circumstances is a function that iterates through a collection, and exits from the iteration by checking for the exception that is thrown when an invalid cursor is used to access elements. When the iteration is complete, the cursor will no longer be valid, and this exception will be thrown. This is not a good programming practice. A function should explicitly test for the cursor being valid. To make this possible, a function must efficiently test this condition (`isValid()`, for the cursor example).

There are situations where the test for a condition can be done more efficiently in combination with performing the actual function. In such cases, it is appropriate, for performance reasons, to make the situation regular (that is, not exceptional) and return the condition as a `Boolean` result. Consider a function that first tests whether an element exists with a given key, and then accesses it if it exists:

```
if (c.containsElementWithKey (key)) {  
    // ...  
    myElement = c.elementWithKey (key); // inefficient  
    // ...  
} else {  
    // ...  
}
```

This solution is inefficient because the element is located twice, once to determine if it is in the collection and once to access it. Consider the following example:

```
try {  
    // ...  
    myElement = c.elementWithKey (key); // bad: exception expected  
    // ...  
} catch (INotContainsKeyException) {  
    // ...  
}
```

This solution is undesirable because an exception is used to change the flow of control of the program. The correct solution is to obtain a cursor together with the containment test, and then to use the cursor for a fast element access:

```
if (c.locateElementWithKey (key, cursor)) {  
    // ...  
    myElement = c.elementAt (cursor); // most efficient  
    // ...  
} else {  
    //...  
}
```

Levels of Exception Checking

Some preconditions are more difficult to check than others. Consider the following possible preconditions:

1. A cursor for a linked collection implementation still points to an element of a given collection.
2. A collection is not empty.

In the production version of a program, it may be less efficient to check the first precondition than the second.

The Collection Class Library provides three levels of precondition checking. They are selected by the following macro variable definitions (use, for example, compile flag `DEF(INO_CHECKS)`):

<code>INO_CHECKS</code>	Check for memory overflow. Other checks may be eliminated to improve performance.
Default	Perform all precondition checks, except the check that a cursor actually points to an element of the collection.
<code>IALL_CHECKS</code>	Perform all precondition checks, including the (costly) check that a cursor actually points to an element of the collection. This extra check can only fail for undefined cursors.

List of Exceptions

The Collection Class Library defines the following exceptions:

ICildAlreadyExistsException

Occurs when you try to add a child to a tree using `addAsChild()` at a position that already contains a child.

ICursorInvalidException

Two cursor properties may lead to the `ICursorInvalidException`:

- Every time a cursor is created, you must specify the collection that it belongs to. If a function takes a cursor as an argument (such as `add()`, `setToFirst()`, and `locate()`), the function can only be applied to the collection that the cursor belongs to. If the function is applied to another collection, the `ICursorInvalidException` results.
- If a function takes a cursor as an input argument (such as `elementAt()`, `removeAt()`, and `replaceAt()`), the cursor must be *valid*. A cursor is valid if it actually refers to some element contained in the collection. You can use the `isValid()` function to determine if a cursor is valid.

IEmptyException

Occurs when a function tries to access an element of an empty collection. Functions that might cause this exception include `firstElement()` and `removeFirstElement()`.

IFullException

Occurs when a function tries to add an element to a bounded collection that is already full. Functions that might cause this exception include `add()` and `addAsFirst()`.

IdenticalCollectionException

Occurs when the function `addAllFrom()` is called with the source collection being the same as the target collection.

InvalidReplacementException

Occurs when, during a `replaceAt()` function, the replacing element has different positioning properties (see “Replacing Elements” on page 93) than the positioning properties of the element to be replaced.

IKeyAlreadyExistsException

Occurs when a function attempts to add an element to a map or sorted map that already has a different element with the same key. Functions that might cause this exception include `add` and `addAllFrom()`.

INotBoundedException

Occurs when the function `maxNumberOfElements()` is applied to a collection that is not bounded.

INotContainsKeyException

Occurs when the function `elementWithKey()` is applied to a collection that does not contain an element with the specified key.

IOutOfMemory

Occurs when a function cannot obtain the space that it requires. This exception is not the result of a precondition violation. Functions that add an element to a collection, including `add()` and `addAsFirst()`, can cause this exception.

IPositionInvalidException

Occurs when a function specifies a position that is not valid in a collection. The functions that might cause this exception include `elementAtPosition()`, `removeAtPosition()`, and `setToPosition()`.

IRootAlreadyExistsException

Occurs when the function `addAsRoot()` is called for a tree that already has a root.

ICollectionResourceException

Occurs when the Collection is constructed and the creation of the internal Resource object fails.

ICollectionLockException

Occurs when an internal lock request fails.

ICollectionUnlockException

Occurs when an internal unlock request fails.

The Hierarchy of Exceptions

In the Collection Class Library, all exceptions are derived from the `ICollectionUnlockException` class described in Chapter 20, “Exception and Trace Classes” on page 215. It provides common functions to access information about an exception that has occurred.

The direct subclasses of `ICollectionUnlockException` used in the Collection Class Library are:

- `ICollectionResourceException`

- `ICollectionLockException`

- `ICollectionUnlockException`

- `ICollectionPreconditionViolation`

- `ICollectionResourceExhausted`

The following figure shows the hierarchy of exceptions:

Exception Hierarchy

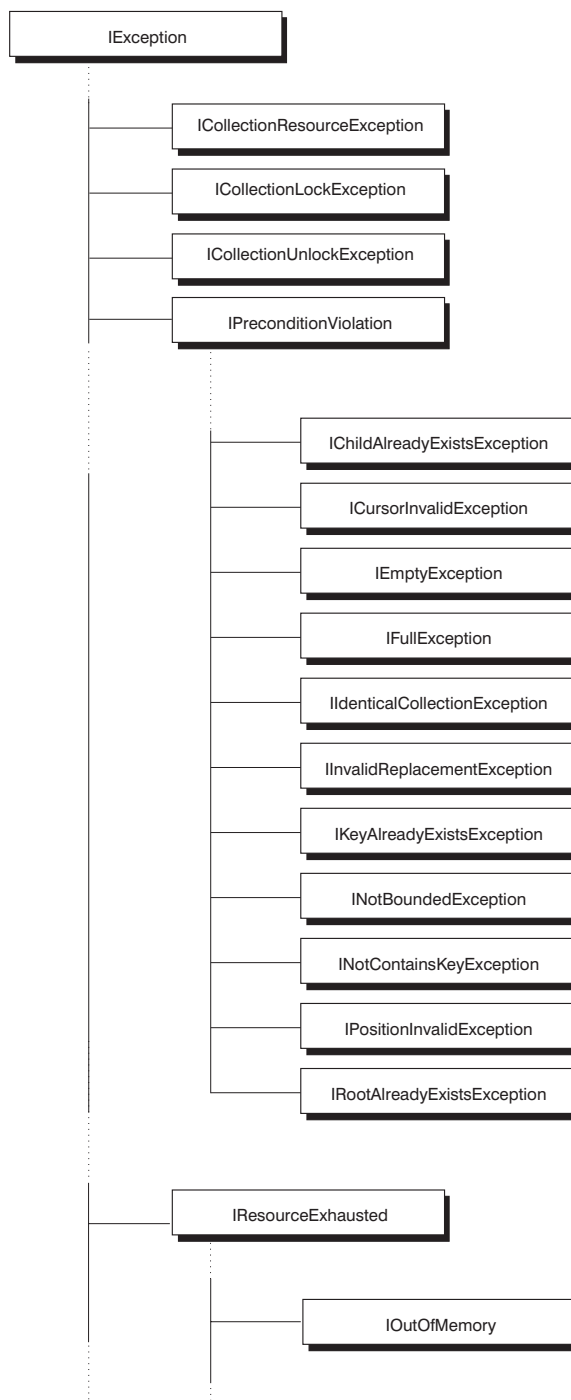


Figure 24. Hierarchy of Exceptions

Chapter 15. Collection Class Library Tutorials

This chapter provides a set of tutorial lessons that you can use to learn common Collection Class Library features. Each lesson builds on the lessons you learned and the library features demonstrated in prior lessons. A section at the end of the chapter describes other tutorials provided with the Collection Class Library that can help you with specific Collection Class Library techniques. Use this chapter if you are beginning to use the library and are unclear on some of the concepts described in earlier chapters of this section.

The lessons in this chapter demonstrate the following capabilities of the Collection Class Library:

- Defining a simple collection
- Adding, removing, and iterating over elements
- Changing the element type
- Changing the collection
- Changing the default implementation

Each lesson has the following format:

- *What the lesson covers:* What you will learn from the lesson.
- *Requirements:* What capabilities must be built into or added to the program.
- *Setup:* What files you will need from previous lessons.
- *Implementation:* Step-by-step instructions for implementing the program requirements. The implementation section includes the required code as well as detailed descriptions of each aspect of the implementation.
- *Source files:* Source file listings showing the contents of, or the order of declaration of functions within, individual source files. Where a source file is not changed from one lesson to the next, it is not listed a second time.
- *Running the program:* A description of what happens when you run the program, observations on the program's behavior, and guidance on optional ways of enhancing or changing the program.
- *What you have learned:* A summary of the Collection Class features that were covered by the lesson.

There are five lessons in this chapter. The following provides an overview of the characteristics of the program used in each lesson, and the Collection Class features the lesson demonstrates:

- | | |
|-----------------|--|
| Lesson 1 | A program that builds a collection of integer elements, and adds three elements to the collection. Nothing is done with the collection after these elements are added, and the program produces no output. This lesson demonstrates how to define the element and collection types with typedefs , how to instantiate a collection, how to add elements to a collection, and how to determine what Collection Class header file to include. |
| Lesson 2 | An enhancement to Lesson 1 that implements a menu so that you can add, list, or remove items, show stock information, or exit the program. Not all these functions are fully implemented at this |

Lesson 1: Defining a Simple Collection

point. The lesson demonstrates how to iterate over a collection and how to remove elements from a collection.

- Lesson 3** An enhancement to Lesson 2 that changes the element type from a built-in type to a class type. The lesson demonstrates how to construct a collection whose elements are of class type, how to determine what element type functions are required, and how to define those functions.
- Lesson 4** In this lesson, you change Lesson 3 to use a different collection. The lesson demonstrates how to choose the correct collection for a given application, how to implement various element and key functions, how to use a cursor to iterate through elements with a given key, and how to count the number of elements with a given key.
- Lesson 5** In this lesson, you change the implementation *variant* of the collection. This does not change the program's external behavior but in real applications changing an implementation variant can affect performance.

Preparing for the Lessons

To set up the lessons, create five data sets beneath the same parent data set, and name them `lesson1` through `lesson5`. You will use these data sets to store the files you create for each lesson.

Compiling the Lessons

You need to create a PDS member to compile the tutorial lessons. Use the data sets for JCL described in “Source Files for the Tutorials” on page 174 as a starting point for creating the JCL for the lessons.

If the compiler produces errors during compilation, check to make sure that you have specified the required library and that you have typed the source code in correctly. Some common errors are misplacing semicolons and failing to close braces or brackets.

Lesson 1: Defining a Simple Collection of Integers

In this lesson, you write a program that builds a very simple collection of integer elements and adds some elements to the collection. This lesson covers the following Collection Class topics:

- Using a **typedef** to define the element type
- Using a **typedef** to define the collection type
- Instantiating the collection
- Adding elements to the collection
- Specifying the Collection Class header file to include

Requirements

The collection must consist of elements of an integer type. The integer type is to be known as type `Bicycle`, so that later lessons can change the members of the type. The program adds three integers to the collection. Their values are unimportant. The collection is to be a bag.

Setup

Change to the `lesson1` data set, and use an editor to create and edit two files:

<code>bike.h</code>	This file will contain declarations and typedefs for the element and collection types.
<code>main.C</code>	This file will contain the <code>main()</code> function.

Implementation

The implementation should use **typedefs** to define the element and collection types, so that if the element or collection type changes later, the changes will be automatically reflected in any code that uses the **typedef**.

Defining the Element Type: Use a **typedef** to define a `Bicycle` as a synonym for an `int`. By using a **typedef**, you make it easier to change the element type later, without having to change anything outside the element's type (or class) definition:

```
// in bike.h
typedef int Bicycle;
```

Notes:

1. In a realistic C++ program using Collection Classes, you do not need to use a **typedef** to define the element type, because it is unlikely that you would switch from a built-in C++ type to a class type.
2. Unless otherwise indicated, you should enter each new block or line of code *below* any code you have already entered.

Defining the Collection Type: Use a **typedef** to define a collection type called `MyCollectionType`. The collection type refers to a bag collection whose elements are `Bicycles`. By using a **typedef**, you make it easier to change the collection type later, without having to change other parts of your code:

```
typedef IBag <Bicycle> MyCollectionType;
```

In this **typedef**, `IBag` is the default implementation for a bag, `Bicycle` is a template argument representing the element type, and `MyCollectionType` is the type name given to the type being defined (a bag of `Bicycle` elements).

Instantiating a Collection: Now that you have defined a **typedef** for both the element and the collection types, you can instantiate a collection with a type specifier and a name:

```
MyCollectionType MyCollection;
```

Place this definition at global scope so that all functions, not only the `main()` function defined in the next step, have access to the collection and its members. Functions other than `main()` are defined in subsequent lessons.

Lesson 1: Defining a Simple Collection

Adding Elements: You can use Chapter 15, “Flat Collection Member Functions” in the *OS/390 C/C++ IBM Open Class Library Reference* to determine what functions you need to use to manipulate elements of a collection. If you consult that chapter, you will find that the `add()` function is the function needed for this lesson. The syntax for `add()` is stated as:

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

For a collection named `MyCollection`, you can add elements using the following syntax:

```
MyCollection.add(aBicycle);
```

Where *aBicycle* is a `Bicycle` (in this case an integer). To add three elements, place code such as the following in `main.C`:

```
void main() {
    Bicycle a,b,c;
    a=458;
    b=12;
    c=365;
    MyCollection.add(a);
    MyCollection.add(b);
    MyCollection.add(c);
}
```

Include Files: Above any **typedefs** or instantiations that use Collection Classes, you must include the header file for any collection you are using. The chapter on bags in the *OS/390 C/C++ IBM Open Class Library Reference* tells you what the header file is for the default implementation of a bag. You should add the following code to the start of `bike.h`, and include `bike.h` in `main.C`:

```
// in bike.h
#include <ibag.h>

// in main.C
#include "bike.h"
```

Source Files for Lesson 1

The files should now contain code similar to the following:

bike.h

```
#include <ibag.h>
typedef int Bicycle;
typedef IBag <Bicycle> MyCollectionType;
MyCollectionType MyCollection;
```

main.C

```
#include "bike.h"

void main() {
    Bicycle a,b,c;
    a=458;
    b=12;
    c=365;
    MyCollection.add(a);
    MyCollection.add(b);
    MyCollection.add(c);
}
```

Running the Program

Compile `main.C` and run the executable. The program does not produce any output, so it appears to do nothing. In fact, it adds three elements to a collection of integers. The collection is lost on program termination. The program is useless in practical terms, but does demonstrate some basic Collection Class concepts. Later lessons build on the code in this lesson, and provide greater functionality, including output of elements.

Chapter 16, “Bag” in the *OS/390 C/C++ IBM Open Class Library Reference* defines a number of element type functions as being required:

- Copy constructor
- Destructor
- Assignment
- Equality test (`operator==`)
- Ordering relation (`operator<`)

You did not have to define these functions in the above example, because for the built-in type `int`, and by extension the user-defined type `Bicycle`, these functions are already defined by the language.

What You Have Learned

This lesson showed you how to define elements and collections using **typedefs**, how to instantiate a collection and elements, and how to add elements to that collection.

Lesson 2: Adding, Listing, and Removing Elements

The first lesson showed you how to create a simple collection and add three elements. This lesson moves the code for adding elements to a separate function, and implements functions for listing and removing elements as well. These functions are called from a main program that dispatches the appropriate function based on the user's choice of a menu option.

This lesson covers the following Collection Class topics:

- Iterating over a collection using applicators (`allElementsDo()`)
- Removing elements from a collection

Requirements

The code in the `main()` function must be replaced by a menu system that gives the user the following options:

1. Add an item
2. List all items
3. Remove an item
4. Show stock information
5. Exit program

Options 1 to 3 must be implemented through functions. Option 5 can be implemented by calling `exit()` or by exiting the scope of the menu selection loop and `main()`. You do not need to implement the function to show stock information in this lesson. Instead, you can implement a function that prints an error message stating that the function is not yet implemented. For all options except the exit

Lesson 2: Adding, Listing, and Removing Elements

option, after the appropriate function returns, the menu should be redisplayed and the user should be able to enter another selection.

Setup

Copy the file `bike.h` from the `lesson1` data set to the `lesson2` data set, and then change your current data set to the `lesson2` data set. You will also create two other files. The three files for this tutorial are:

<code>bike.h</code>	Contains the element and collection typedefs.
<code>lesson.C</code>	Contains functions for adding, removing, listing, and showing stock information on items.
<code>main.C</code>	Contains the main menu for the program.

Implementation

You need to replace the body of the `main()` function with the menu handling and function dispatching code. You will make use of I/O Stream input and output to implement the functions that add, list, or remove items. One advantage of using the I/O Stream classes instead of functions like `printf()` and `scanf()` is that, when the element type is changed, you can define input and output operators for the type, and the I/O Stream input and output functions will continue to work without change.

Including the `iostream.h` Header File: You should include `iostream.h` at the start of `lesson.C` so that you can use the `cin`, `cout`, and `cerr` streams that are predefined by the `iostream` class. You should also include the header file `bike.h` so that you can access the `Bicycle` class and associated functions.

```
#include <iostream.h>
#include "bike.h"
```

Adding Items: Before the definition of `main()`, define a function `addItem()` that requests user input for the item, then adds the item to the collection. The item is added using the `add()` function described in the first lesson. Here is one way to implement such a function:

```
// in lesson.C
void addItem() {
    Bicycle tbike;
    cout << "Enter item: ";
    cin >> tbike;
    while (cin.fail()) {
        cin.clear();
        cin.ignore(1000, '\n');
        cerr << "Input error, please re-enter: ";
        cin >> tbike;
    }
    MyCollection.add(tbike);
}
```

Note: You should also add a declaration for this and subsequent functions in `main.C`.

The function uses a temporary `Bicycle` object to contain the input until the element is copied into the collection. The function displays a prompt, reads input, and tests for valid input. The `while (cin.fail())` block clears any input errors and asks for

input again. Once the element is successfully read from input, it is added to the collection.

Because `tbike` is actually an `int` in the current version, an operator `>>` is already defined for it. Later, when you change the `Bicycle` type to a user-defined class, you will have to add an operator `>>` for that class.

Listing Items: Before you can list all items, you must define a function that prints a single item. This function can then be invoked by the `allElementsDo()` member function of `MyCollection`. (`allElementsDo()` is described in “allElementsDo” in the *OS/390 C/C++ IBM Open Class Library Reference*.) Any function invoked by `allElementsDo()` must have a return type of `IBoollean`, and must have two arguments: a **const** reference to the argument and a pointer to void. The pointer to void is used to pass additional arguments to the applied function, if required by the function. For the printing function in this lesson you do not need to pass additional arguments, because the function does not use them. In such cases you pass a `void*` second argument:

```
// in lesson.C
IBoollean printItem (Bicycle const& bike, void* /* Not used */) {
    cout << bike << endl;
    return true;
}
```

The `printItem()` function should always return `true` because it should display the value of each element of the collection. If you wanted certain values of elements to cause printing to halt, you would have the function return `false` for any such element. A return value of `false` causes the `allElementsDo()` function to stop iterating over the collection.

Just as there was no need to define an input operator for `Bicycle`, there is no need to define an output operator either, as long as `Bicycle` represents an `int`.

Now define the function `listItems()` to call the `printItem()` function for each element of the collection. Use the `allElementsDo()` function for the collection, and use the `printItem()` function as argument. `allElementsDo()` then calls the function for every element of the collection.

```
// in lesson.C
void listItems() {
    MyCollection.allElementsDo( printItem );
}
```

Removing Items: To remove an element from a collection, you need to use the `remove()` member function. This function is described in Chapter 15, “Flat Collection Member Functions” in the *OS/390 C/C++ IBM Open Class Library Reference*. `remove()` returns `true` if the element was found in the collection and was removed, or it returns `false` if the element was not found in the collection. Your removal function should print an error if the element is not successfully removed. In the version below, the condition that determines whether removal was successful actually invokes the `remove()` function:

```
// in lesson.C
void removeItem() {
    Bicycle tbike;
    cout << "Enter item to remove: ";
    cin >> tbike;
```

Lesson 2: Adding, Listing, and Removing Elements

```
while (cin.fail()) {
    cin.clear();
    cin.ignore(1000, '\n');
    cerr << "Input error, please re-enter: ";
    cin >> tbike;
}
if (!MyCollection.remove(tbike))
    cerr << "Item not found!\n";
}
```

Showing Stock Information: For now, you can define this function to display an error message without changing the collection:

```
// in lesson.C
void showStock() {
    cerr << "Function not implemented yet!\n";
}
```

Main Menu: Finally, change the code in `main()` to display the menu items, accept input, and take appropriate action. Because this code will remain relatively unchanged for subsequent lessons, place it in a separate file, `main.C`, and include `lesson.C` before the code of `main()`. A possible version of `main.C` is shown below.

Source Files for Lesson 2

You should have two source files defined at this point. Their names and sample contents are:

main.C

```
#include <iostream.h>
#include <stdlib.h> // for use of exit() function
void addItem(), listItems(), showStock(), removeItem();

void main() {
    enum Choices { Add, List, Stock, Remove, Exit };
    int menuChoice=0;
    char* menu[5] = {"Add an item",
                    "List items",
                    "Show stock information",
                    "Remove an item",
                    "Exit" };
    while (menuChoice!=5) {
        cout << "\n\nSimple Stock Management System\n\n";
        for (int i=0; i<5; i++)
            cout << i+1 << ". " << menu[i] << '\n';
        cout << "\nEnter a selection (1-5): ";
        cin >> menuChoice;
        while (cin.fail()) {
            // get input again if nonnumeric was entered
            cin.clear();
            cin.ignore(1000, '\n');
            cerr << "Enter a selection between 1 and 5!\n";
            cin >> menuChoice;
        }
        switch (menuChoice) {
            case 1: addItem(); break;
            case 2: listItems(); break;
            case 3: showStock(); break;
            case 4: removeItem(); break;
            case 5: exit(0);
            default: cerr << "Enter a selection between 1 and 5!\n";
        }
    }
}
```

lesson.C

```

// lesson.C
#include <iostream.h>
#include <ibag.h>
#include "bike.h"

void addItem() {
    Bicycle tbike;
    cout << "Enter item: ";
    cin >> tbike;
    while (cin.fail()) {
        cin.clear();
        cin.ignore(1000, '\n');
        cerr << "Input error, please re-enter: ";
        cin >> tbike;
    }
    MyCollection.add(tbike);
}

IBoolen printItem (Bicycle const& bike, void* /* Not used */) {
    cout << bike << endl;
    return true;
}

void listItems() {
    MyCollection.allElementsDo( printItem );
}

void removeItem() {
    Bicycle tbike;
    cout << "Enter item to remove: ";
    cin >> tbike;
    while (cin.fail()) {
        cin.clear();
        cin.ignore(1000, '\n');
        cerr << "Input error, please re-enter: ";
        cin >> tbike;
    }
    if (!MyCollection.remove(tbike))
        cerr << "Item not found!\n";
}

void showStock() {
    cerr << "Function not implemented yet!\n";
}

```

Running the Program

Compile main.C and lesson.C, link them, and run the program. You can enter elements into the collection, list the elements, remove them, or exit from the program. If you select the option to display stock information, an error message is displayed and no action is taken.

Elements appear to be ordered: If you enter more than one integer into the collection, and then list the collection's elements, you may find that the collection has been sorted from the smallest to the largest element. Do not rely on this ordering relation, because a Bag is an unordered, unsorted collection, and changes to your code or to the Collection Class Library could change the order in which elements are accessed.

Multiple equal elements are supported: If you add the number 7 to the collection three times and list the items, the number 7 appears three times. If you then remove the number 7 once, the number 7 still appears twice. A bag supports multiple equal elements.

What You Have Learned

This lesson showed you how to use the `allElementsDo()` function to iterate over elements of a collection, and how to provide a function to `allElementsDo()` that is called for each iterated element. The lesson also demonstrated how to use the `remove()` function to remove elements from a collection.

Lesson 3: Changing the Element Type

Now that you have a working program that allows you to add, list, or remove elements from a collection, you are ready to change the element type to something more complex than an integer.

This lesson covers the following Collection Class topics:

- Defining an element type as a class
- Determining what element type functions are required
- Defining those element type functions

Requirements

The element type must be changed from the built-in integer type to a class type with the following data members:

- A string representing the manufacturer or make of the bicycle
- A string representing the model of the bicycle
- An integer representing the type of bicycle: racing, touring, or mountain bike
- An integer representing the price of the bicycle

Setup

Copy the files `bike.h`, `lesson.C`, and `main.C` from the `lesson2` data set to the `lesson3` data set, and then change your current data set to the `lesson3` data set. Use an editor to modify these files, and to create a new file `bike.C`, which will contain function definitions for functions declared in `bike.h`.

Implementation

First move the **`typedef`** definition for the collection and the **`#include`** statement for `ibag.h` from `bike.h` to `lesson.C`, where they are actually made use of.

You can use the `IString` class to handle the strings for `make` and `model`. This class includes operators for element equality, ordering relation, and addition (concatenation), all of which will be used in this or later lessons.

Defining the Element Type: In keeping with good object-oriented programming practice, you should separate the member function definitions from the class definition, by placing the class definition in `bike.h` and the definitions of member functions in `bike.C`. You should compile each `.C` file separately, and link them together.

Class Data Members: The following code defines the data members of `Bicycle`. You should replace the **`typedef`** for the element with the declaration for class `Bicycle`. Two header files are also included because they are required by members of the class. Place the following code in `bike.h`.


```
#include <istring.hpp> // access to IString class
#include <iostream.h> // access to iostream class

class Bicycle {
public:
    IString Make;
    IString Model;
    int Type;
    int Price;
    // ... Member functions to be declared later and defined in bike.C
};
```

The following code defines an enumerator (used to determine the type of bicycle) and an array of IString objects (used to display the types of bicycle). Place it in bike.C:

```
enum bikeTypes { Racing, Touring, MountainBike };
IString btype[3]={ "Racing", "Touring", "Mountain Bike"};
```

Selecting What Element Type Functions to Implement: When you implement the element type as a user-defined type (a class), you must define certain element functions, and in some cases key-type functions, for that element. These functions are used by Collection Class functions to locate, add, copy, remove, sort, or order elements within their collection, and to determine whether two elements of a collection are equal. For example, you may need to define element equality through an `operator==`, so that Collection Class functions can determine whether an element you try to add to the collection is identical to an element already present in the collection. Provided you use the correct return type and calling arguments, there is no right or wrong way to code many of these functions. An equality function for elements consisting of two `int` data members, for example, could return `true` (meaning that two elements are equal) if the *difference* between the two data members is the same for both elements. In this case, the objects (3,8) and (4,9) would be equal.

To determine what element and key-type functions you need to implement for a given collection, you should consult the appropriate collection's chapter in the *OS/390 C/C++ IBM Open Class Library Reference*. For this lesson, the collection is a bag. When you are first developing a program, you should use the default implementation of the collection, which is always the first implementation variant listed under the chapter's "Template Arguments and Required Functions" section. For each implementation variant, a list of required functions is provided, and you must either implement these functions for your element class, or determine that they are automatically generated by the compiler. In the case of the default implementation of a Bag, the following required functions are shown, under the heading "Element Type":

- Copy constructor
- Destructor
- Assignment
- Equality test
- Ordering relation

For this lesson, you also need to implement input and output operators and a default constructor (used by the input operator and other functions).

Lesson 3: Changing the Element Type

Default Constructor: The default constructor should initialize all data members to blank strings or zero integers:

```
// in bike.h, within class declaration
Bicycle() : Make(""), Model(""), Type(0), Price(0) {}
```

Assignment Operator and Destructor: There is no need to define these explicitly. The compiler generates a default assignment operator and destructor that are suitable for the program.

Copy Constructor: This function is used by the Collection Classes and by the input operator. Declare and define it as follows:

```
// in bike.h:
Bicycle(IString mk, IString md, int tp, int pr) :
    Make(mk), Model(md), Type(tp), Price(pr) {}
```

Equality Test: The equality test (operator==) should return true if two bicycles have the same make and model, and false if not:

```
// in bike.h:
IBoolean operator== (Bicycle const& b) const;

// in bike.C:
IBoolean Bicycle::operator== (Bicycle const& b) const
{ return ((Model==b.Model) && (Make==b.Make)); }
```

Ordering Relation: The ordering relation (operator<) should indicate whether the first bicycle would appear before or after the second bicycle in an alphabetically sorted list:

```
// in bike.h:
IBoolean operator< (Bicycle const& b) const;

// in bike.C:
IBoolean Bicycle::operator< (Bicycle const& b) const
{ return ((Make<b.Make) || (Make==b.Make && Model<b.Model)); }
```

You can use the < and == operators for IString objects because they are defined for the IString class to indicate alphanumeric sorting order.

Input Operator: This operator is required by the addItem() and removeItem() functions defined previously. Both this and the output operator are declared *outside* the class definition, at the bottom of bike.h, and they are defined in bike.C. The input operator stores the alphanumeric data members of Bicycle in char arrays to avoid the overhead of constructing temporary IString objects.

```
// in bike.h:
istream& operator>> (istream& is, Bicycle& bike);

// in bike.C:
istream& operator>> (istream& is, Bicycle& bike) {
    char make[40], model[40];
    char typeChoice;
    float price;
    int type=-1;
    cin.ignore(1, '\n'); // ignore linefeed from previous input
    cout << "\nManufacturer: ";
    cin.getline(make, 40, '\n');
    cout << "Model: ";
    cin.getline(model, 40, '\n');
```

```

while (type == -1) {
    cout << "Racing, Touring, or Mountain Bike (R/T/M): ";
    while (cin.fail()) {
        cin.clear();
        cin.ignore(1000, '\n');
        cerr << "Racing, Touring, or Mountain Bike (R/T/M): ";
        cin >> typeChoice;
    }
    switch (typeChoice) {
        case 'r':
        case 'R': { type=Racing; break; }
        case 't':
        case 'T': { type=Touring; break; }
        case 'm':
        case 'M': { type=MountainBike; break; }
        default: { cerr << "Incorrect type, please re-enter\n"; }
    }
}
cout << "Price ($$. $$): ";
cin >> price;
while (cin.fail()) {
    cin.clear();
    cin.ignore(1000, '\n');
    cerr << "Enter a numeric value: ";
    cin >> price;
}
price*=100;
bike=Bicycle(make,model,type,price);
return is;
}

```

Output Operator: The output operator is required by the `listItems()` function, and may later be required by other functions. It should display the make, model, type, and price of a bicycle:

```

// in bike.h:
ostream& operator<< (ostream& os, Bicycle bike);

// in bike.C:
ostream& operator<< (ostream& os, Bicycle bike) {
    return os << bike.Make
        << "\t" << bike.Model
        << "\t" << btype[bike.Type]
        << "\t" << float(bike.Price)/100;
}

```

Source Files for Lesson 3

The program should now be placed in the following files. Some function bodies have been replaced with ellipses for brevity. `main.C` remains unchanged and is not shown.

lesson.C

```

// lesson.C
#include <iostream.h>
#include <ibag.h>
#include "bike.h"
typedef IBag<Bicycle> MyCollectionType;
MyCollectionType MyCollection;

void addItem() { /* ... */ }
IBoolean printItem (Bicycle const& bike, void* /* Not used */)
{ /* ... */ }
void listItems() { /* ... */ }
void removeItem() { /* ... */ }
void showStock() { /* ... */ }

```

bike.h

Lesson 3: Changing the Element Type

```
#include <istring.hpp> // access to IString class
#include <iostream.h> // access to iostream class

class Bicycle {
public:
    IString Make;
    IString Model;
    int Type;
    int Price;
    Bicycle() : Make(""), Model(""), Type(0), Price(0) {}
    Bicycle(IString mk, IString md, int tp, int pr) :
        Make(mk), Model(md), Type(tp), Price(pr) {}
    IBoolean operator== (Bicycle const& b) const;
    IBoolean operator< (Bicycle const& b) const;
};
istream& operator>> (istream& is, Bicycle& bike);
ostream& operator<< (ostream& os, Bicycle bike);
```

bike.C

```
#include <istring.hpp>
#include "bike.h"
enum bikeTypes { Racing, Touring, MountainBike };
IString btype[3]={ "Racing", "Touring", "Mountain Bike"};

IBoolean Bicycle::operator== (Bicycle const& b) const
{ return ((Model==b.Model) && (Make==b.Make)); }

IBoolean Bicycle::operator< (Bicycle const& b) const
{ return ((Make<b.Make) || (Make==b.Make && Model<b.Model)); }

istream& operator>> (istream& is, Bicycle& bike) {
    char make[40], model[40];
    char typeChoice;
    float price;
    int type=-1;
    cin.ignore(1, '\n'); // ignore linefeed from previous input
    cout << "\nManufacturer: ";
    cin.getline(make, 40, '\n');
    cout << "Model: ";
    cin.getline(model, 40, '\n');
    while (type == -1) {
        cout << "Racing, Touring, or Mountain Bike (R/T/M): ";
        cin >> typeChoice;
        while (cin.fail()) {
            cin.clear();
            cin.ignore(1000, '\n');
            cerr << "Racing, Touring, or Mountain Bike (R/T/M): ";
            cin >> typeChoice;
        }
        switch (typeChoice) {
            case 'r':
            case 'R': { type=Racing; break; }
            case 't':
            case 'T': { type=Touring; break; }
            case 'm':
            case 'M': { type=MountainBike; break; }
            default: { cerr << "Incorrect type, please re-enter\n"; }
        }
    }
    cout << "Price ($$. $$): ";
    cin >> price;
    while (cin.fail()) {
        cin.clear();
        cin.ignore(1000, '\n');
        cerr << "Enter a numeric value: ";
        cin >> price;
    }
    price*=100;
    bike=Bicycle(make,model,type,price);
    return is;
}
```

```
ostream& operator<< (ostream& os, Bicycle bike) {
    return os << bike.Make
        << "\t" << bike.Model
        << "\t" << btype[bike.Type]
        << "\t" << float(bike.Price)/100;
}
```

Running the Program

Compile and link `bike.C`, `main.C` and `lesson.C`, and then run the program.

If you add two bicycles with the same make and model, but different types or prices, the second bicycle's entry will be identical to the first when the bicycles are listed. The reason is that element equality is defined only in terms of the make and model. When you add what the collection considers to be an equal element, the existing element is duplicated by the `add()` function.

When you remove an item, the input operator asks you to enter all fields for the item to remove. Again, because element equality is defined only for the make and model fields, the information you provide for bicycle type and price is not used in determining which element to remove. If you define a bicycle:

```
Smithson    37Q    Racing    $270.00
```

You can remove that bicycle's entry by removing:

```
Smithson    37Q    Mountain Bike    $399.99
```

These limitations will be corrected in the next lesson.

What You Have Learned

In this lesson, you moved from using built-in types as elements of a collection to using user-defined or class types. When you create a collection using class-type elements, you must define certain element functions. This lesson showed you how to determine what element functions are required, and how to implement them.

Lesson 4: Changing the Collection

When you design an actual application using the Collection Class Library, you should choose the collection best suited to your program at the design stage. Nevertheless, requirements may change, and if you have followed the techniques used in this lesson such as specifying the collection type with a **typedef**, you can change the collection type without having to rewrite the entire application. Only minor changes are required to existing code, and a few simple element or key-type functions may need to be added or changed.

This section illustrates the following Collection Class concepts:

- Selecting the correct collection type
- Implementing a key
- Defining key access
- Defining key equality
- Defining a key hash
- Using a cursor to iterate through elements with a given key
- Counting the number of elements of a given key

Lesson 4: Changing the Collection

Requirements

The program should be changed so that two bicycles of the same model and make can have different type and price information. When users asks to delete a bicycle, they should not have to enter the bicycle and price information; instead, a list of all bicycles of the specified make and model should be displayed, and the user should be able to select which bicycle to remove from the collection. The `showStock()` function should also be implemented, so that it shows the number of a given make and model of bicycle currently in the collection.

Setup

Copy the files `bike.h`, `bike.C`, `lesson.C`, and `main.C` from the `lesson3` data set to the `lesson4` data set, and then change your current data set to the `lesson4` data set. Use an editor to modify the files as described below.

Implementation

The collection must have the following characteristics:

Key access, so that an element can be accessed using only its make and model information (for the listing and removing functions)

No element order, because order is not specified as a requirement

Multiple elements with the same key, so that several bicycles of the same make and model can be present in the collection

Element equality, so that elements with the same make and model can have different price and type information

You can use Figure 8 on page 78 to determine what collection best meets the requirements listed above. Begin by applying one requirement to the figure to narrow down the number of possible collections. Apply a second requirement to the remainder, and continue until you have found all valid collections. In this example, there is one valid collection, selected as follows:

- Elements have a key (the make and model). This means that any of the following collections may be a candidate:
 - Map
 - Relation
 - Sorted map
 - Sorted relation
 - Key set
 - Key bag
 - Key sorted set
 - Key sorted bag
- The order of elements is not important. This means that all sorted collections can be removed from the list above, leaving:
 - Map
 - Relation
 - Key set
 - Key bag
- Multiple elements may have the same key. This leaves relation and key bag.
- Element equality is required, so that individual elements with the same key can be distinguished. This leaves relation.

A relation differs from a bag in that it is instantiated using a key type as well as the element type, and requires the following additional functions:

Element type: Key access

Key type: Equality test and hash function.

These functions are defined below.

Changing the Collection Type Definition: Before you redefine the functions in `Lesson.C`, you need to change the include file and **typedef** for the collection type so that they use relation instead of bag:

```
// lesson.C
#include <irel.h> // was ibag.h
//...
typedef IRelation<Bicycle, IString> MyCollectionType;
// was typedef IBag<Bicycle> MyCollectionType;
```

Notice that `IRelation` takes two template arguments, an element type and a key type. All collections that have a key must be defined with a template argument for key type as well as one for element type.

Ordering Relation: A relation does not require an operator for ordering relation (`operator<`). You defined this operator when the collection was implemented as a bag. You should comment it out or remove it for this implementation. This function is declared in `bike.h` and defined in `bike.C`.

Implementing a Key: The key consists of the make and model of the bicycle. You can use an `IString` to implement the key. Because the return value of the `key()` function must be a **const** reference, and because the `key()` function cannot change the element, the key must be determined before the `key()` function is called. The logical place to do this is in the element constructor (in `bike.h`), because the overhead of generating the key only occurs once per element. You can add a key data member to the collection, and have it initialized when the copy constructor is called. In the example below, the key is named `MMKey` (which stands for Make/Model Key):

```
// in bike.h:
class Bicycle {
    IString MMKey; // add a private data member for the key
public:
    // public data members and member functions
    Bicycle(IString mk, IString md, int tp, int pr);
    // ...
};

// in bike.C:
Bicycle::Bicycle(IString mk, IString md, int tp, int pr) :
    Make(mk), Model(md), Type(tp), Price(pr),
    MMKey(mk+md) {}
```

Defining Key Access: The key access function must be defined *outside* of the element class. It has one argument, whose type is the element type. The key access function must call a member function that returns the key, in this case a function named `getKey()`. (The actual name does not matter.) The member function accesses the private data member `MMKey`.

Lesson 4: Changing the Collection

```
// in bike.h:
class Bicycle {
    IString MMKey;
public: // ... data members and member functions
    IString const& getKey() const;
};

inline IString const& key (Bicycle const& bike)
{ return bike.getKey(); }

// in bike.C:
IString const& Bicycle::getKey() const { return MMKey; }
```

The key access function must be declared with the name `key()`, with a **const** reference to the key as its return value, and a **const** reference to the element as its argument.

Equality Test: Equality for elements should be defined such that the key (that is, the make and model), the type, and the price are the same for two bicycles. The `operator==` function in `bike.C` can be redefined as follows:

```
IBoolean Bicycle::operator== (Bicycle const&b) const {
    return (MMKey==b.MMKey && Type==b.Type && Price==b.Price);
}
```

Key Hash Function: The hash function provides a shortcut for Collection Class search functions to find matches to a key. The search functions first call the hash function on a key for which they need to locate an element. They use the hash value returned to look for matches to that hash in a hash table. They then use the full key to determine which of the hash function's matches have the correct key. The hash key-type function is not a member function of the element's class. It is called by the searching function, with a key argument (the key on which to derive the hash) and an unsigned long (the maximum hash value). The return value is the hash, and it cannot exceed the maximum hash value. The hash function should be defined in `lesson.C` and must have the following return type and parameters:

```
unsigned long hash ( IString const& keyName,    unsigned long hashInput );
```

You can define the hash using the hashing function provided in `istdops.h` for `char*` values:

```
unsigned long hash (IString const &aKey, unsigned long hashInput) {
    return hash( (const char*)aKey, hashInput);
}
```

Using Cursors to Remove Items: A Collection Class cursor (not related to the cursor used to move about a cursor screen) is a reference to an element in a collection. For an overview of cursors, see "Cursors" on page 93.

The `removeItem()` function must be redefined so that it requests the make and model of bicycle to remove, lists all matching bicycles, and lets the user choose which match to remove. Once matching bicycles have been displayed, a cursor can be used to locate the bicycle the user wishes to delete. The cursor is defined as follows, immediately after the collection `MyCollection` is declared, in `lesson.C`:

```
MyCollectionType::Cursor thisOne (MyCollection);
```

After the user enters a make and model to search for, the `removeItem()` function should iterate through all elements that match the key, by using `locateElementWithKey()` to find the first matching element, and

`locateNextElementWithKey()` to find all subsequent matching elements. Both these functions require a cursor as their second argument, and the cursor points to the located element when the functions return. The first part of `removeItem()` can be redefined as follows:

```
void removeItem() {
    Bicycle tbike;
    int choice, cursct=1;
    cout << "\nRemove an item";
    cin >> tbike;
    if (MyCollection.numberOfElementsWithKey(tbike.getKey()) > 0) {
        MyCollection.locateElementWithKey(tbike.getKey(), thisOne);
        cout << cursct << ". " << MyCollection.elementAt(thisOne) << endl;
        for ( cursct=2;
            MyCollection.locateNextElementWithKey(
                tbike.getKey(), thisOne);
            cursct++)
        { cout << cursct << ". "
            << MyCollection.elementAt(thisOne) << endl; }
        //... Remainder to be defined later
    }
}
```

In the above fragment, the user is asked for a bicycle make and model to remove. If any elements match the make and model (this is determined by testing the `numberOfElementsWithKey()` function for a nonzero return), all such elements are located by key. The `locateElementWithKey()` function sets its cursor to point to the first matching element, and the `locateNextElementWithKey()` function advances the cursor from the current match to the next match in the collection. The elements are accessed for output using the `elementAt()` function, which returns a reference to the element pointed to by the cursor argument.

Once the matching elements have been displayed with a number beside each one, the program should ask the user to enter a number matching the number of the element to remove. The matching elements can then be iterated over again until the number of elements iterated over matches the user's selection, and the element pointed to by the cursor is then deleted. The following code excerpt is the second part of the `removeItem()` function:

```
// Insert this at "...Remainder to be defined later" in removeItem().
cout << "\nEnter item to remove, or 0 to return: ";
cin >> choice;
if (choice<=0 || choice > cursct) return;
MyCollection.locateElementWithKey(tbike.getKey(),thisOne);
    // locate the first matching element again
for ( cursct=2;
    cursct<=choice &&
    MyCollection.locateNextElementWithKey    // check for valid
        (tbike.getKey(), thisOne);    // next match
    cursct++)
    ; // null loop - header contains the code to be executed
MyCollection.removeAt(thisOne);
}
else
    cerr << "No bicycles of this make and model were found.\n";

// The closing brace below was already part of removeItem().
// Do not duplicate it.
}
```

Note: The `locateNextElementWithKey()` function invalidates the cursor if it cannot find a next element with the key provided. An invalidated cursor does not point to any element of the collection. Some flat collection member functions that use cursors require that the cursor be valid (`locateNextElementWithKey()` is one such function). Before you use a cursor with such a function, you need to validate the

Lesson 4: Changing the Collection

cursor by using a function that takes a cursor as argument but does not require a valid cursor on entry. `locateElementWithKey()` is one such function.

In both excerpts of `removeItem()` above, the elements with matching keys are iterated over by code in the header of the loop. In the second case, the loop has no body. You can use this coding style because all the `locate...` functions have a return type of `IBoollean`, which can be used in condition tests such as those in loop control expressions.

Showing Stock Information: `showStock()` must be rewritten so that, for a given make and model, it displays the number of matching elements in the collection. The `numberOfElementsWithKey()` function can be used:

```
void showStock() {
    Bicycle tbike;
    int count;
    cout << "Stock information for a model";
    cin >> tbike;
    count=MyCollection.numberOfElementsWithKey(tbike.getKey());
    if (count!=1)
        cout << "Currently there are " << count << " bicycles ";
    else
        cout << "Currently there is 1 bicycle ";
    cout << "of this make and model in stock." << endl;
}
```

Changing the Input Operator and `addItem()`: As the program now stands, the input operator requests input for all data members of `Bicycle`, including type and price information. This means that, when you select an item to remove or to show stock information on, you must specify type and price information even though this information is ignored. Therefore you need to move the request for type and price information out of the operator `>>` definition in `bike.C` and into `addItem()`, so that the user only needs to enter type and price information when an item is being added to the collection. You also need to add the enumeration `bikeTypes` to `lesson.C` so that `addItem()` has access to them.

See the “Source Files” section below for the changes required to `addItem()` and operator `>>`.

Source Files for Lesson 4

The main program in `main.C` has not been changed. The following excerpts show the layout of code between `lesson.C` and `bike.h`. Function bodies that remain unchanged from the preceding lesson have been replaced by ellipses.

bike.h

```
#include <istring.hpp> // access to IString class
#include <iostream.h> // access to iostream class

class Bicycle {
    IString MMKey;
public:
    IString Make;
    IString Model;
    int Type;
    int Price;
    Bicycle();
    Bicycle(IString mk, IString md, int tp, int pr);
    IBoolean operator== (Bicycle const& b) const;
//    IBoolean operator< (Bicycle const& b) const;
    IString const& getKey() const;
};
```

```
inline IString const& key (Bicycle const& bike)
{ return bike.getKey(); }

istream& operator>> (istream& is, Bicycle& bike);
ostream& operator<< (ostream& os, Bicycle bike);
```

bike.C

```
#include <istring.hpp>
#include "bike.h"
enum bikeTypes { Racing, Touring, MountainBike };
IString btype[3]={ "Racing", "Touring", "Mountain Bike"};

Bicycle::Bicycle() : Make(""), Model(""), Type(0), Price(0) {}
Bicycle::Bicycle(IString mk, IString md, int tp, int pr) {
    Make=mk;
    Model=md;
    Type=tp;
    Price=pr;
    MMKey=Make+Model;
}

// Comment out the ordering relation operator
// IBoolean Bicycle::operator< (Bicycle const& b) const
// { return ((Make<b.Make) || (Make==b.Make && Model<b.Model)); }
IBoolean Bicycle::operator== (Bicycle const&b) const {
    return (MMKey==b.MMKey && Type==b.Type && Price==b.Price);
}
IString const& Bicycle::getKey() const { return MMKey; }

istream& operator>> (istream& is, Bicycle& bike) {
    char make[40], model[40];
    char typeChoice;
    float price=0;
    int type=-1;
    cin.ignore(1, '\n'); // ignore linefeed from previous input
    cout << "\nManufacturer: ";
    cin.getline(make, 40, '\n');
    cout << "Model: ";
    cin.getline(model, 40, '\n');
    bike=Bicycle(make,model,type,price);
    return is;
}

ostream& operator<< (ostream& os, Bicycle bike) { /* ... */ } // unchanged
```

lesson.C

```
// lesson.C
#include <iostream.h>
#include <irel.h> // was ibag.h
#include "bike.h"
enum bikeTypes { Racing, Touring, MountainBike };
typedef IRelation<Bicycle,IString> MyCollectionType;

MyCollectionType MyCollection;
MyCollectionType::Cursor thisOne (MyCollection);

IBoolean printItem (Bicycle const& bike, void* /* Not used */)
{ /* ... */ }

void addItem() {
    Bicycle tbike;
    char typeChoice;
    float price;
    int type=-1;
    cout << "Enter item: ";
    cin >> tbike;
    while (type == -1) {
        cout << "Racing, Touring, or Mountain Bike (R/T/M):";
        cin >> typeChoice;
```

Lesson 4: Changing the Collection

```
while (cin.fail()) {
    cin.clear();
    cin.ignore(1000, '\n');
    cerr << "Racing, Touring, or Mountain Bike (R/T/M): ";
    cin >> typeChoice;
}
switch (typeChoice) {
    case 'r':
    case 'R': { type=Racing; break; }
    case 't':
    case 'T': { type=Touring; break; }
    case 'm':
    case 'M': { type=MountainBike; break; }
    default: { cerr << "Incorrect type, please re-enter\n"; }
}
}
cout << "Price ($$. $$): ";
cin >> price;
price*=100;
tbike.Type=type;
tbike.Price=price;
MyCollection.add(tbike);
}

void listItems() { /* ... */ }
void removeItem() {
    Bicycle tbike;
    int choice, cursct=1;
    cout << "\nRemove an item";
    cin >> tbike;
    if (MyCollection.numberOfElementsWithKey(tbike.getKey()) > 0) {
        MyCollection.locateElementWithKey(tbike.getKey(), thisOne);
        cout << cursct << ". " << MyCollection.elementAt(thisOne) << '\n';
        for ( cursct=2;
            MyCollection.locateNextElementWithKey(
                tbike.getKey(), thisOne);
            cursct++)
        { cout << cursct << ". "
            << MyCollection.elementAt(thisOne) << '\n'; }
        cout << "\nEnter item to remove, or 0 to return: ";
        cin >> choice;
        if (choice<=0 || choice > cursct) return;
        MyCollection.locateElementWithKey(tbike.getKey(), thisOne);
        // locate the first matching element again
        for ( cursct=2;
            cursct<=choice &&
            MyCollection.locateNextElementWithKey // check for valid
                (tbike.getKey(), thisOne); // next match
            cursct++)
        ; // null loop - header contains the code to be executed
        MyCollection.removeAt(thisOne);
    }
    else
        cerr << "No bicycles of this make and model were found.\n";
}

void showStock() {
    Bicycle tbike;
    int count;
    cout << "Stock information for a model";
    cin >> tbike;
    count=MyCollection.numberOfElementsWithKey(tbike.getKey());
    if (count!=1)
        cout << "Currently there are " << count << " bicycles ";
    else
        cout << "Currently there is 1 bicycle ";
    cout << " of this make and model in stock." << endl;
}

unsigned long hash (IString const &aKey, unsigned long hashInput) {
    return hash( (const char*)aKey, hashInput);
}
```

Running the Program

You can enter multiple bicycles of the same make and model, with different price or type information, and all such models will appear when you select the “List items” option. When you ask for stock information, the program displays the number of elements in the collection that match the make and model information you specify. When you remove an item, the program asks you for the make and model, displays a list of matching items, and lets you specify which item to remove. The program removes that item.

What You Have Learned

The Collection Class Library offers a wide range of collections with different characteristics. In this lesson, you learned how to select an appropriate collection based on the characteristics of the data being placed in the collection and on the intended uses of the data. Many Collection Classes are accessed or sorted using a key, and you learned how to define key access, equality, and hash functions, and how to iterate through a key collection using a key cursor.

Lesson 5: Changing the Implementation Variant

You should pursue changing the default implementation to an implementation variant only after the program is functionally complete and has been fully debugged. The purpose of changing to a nondefault implementation variant is to improve performance. This lesson shows you how to change the code defined in “Lesson 3: Changing the Element Type” on page 158 so that it is functionally equivalent, but uses `IBagAsDilutedTable` rather than `IBag`. The lesson assumes that you have done some analysis of your code, and have determined that this implementation variant may provide better performance. In the case of a full-fledged application, once you change the implementation variant, you should compile the program and time it against the original implementation to determine whether there is a worthwhile gain in performance.

This section illustrates the following Collection Class concepts:

- Changing the implementation variant header file
- Changing the implementation variant template and template arguments
- Determining what functions are required by the implementation variant

Requirements

The only implementation variant for a relation is the variant that allows you to use a generic operations class.

If the collection were still a bag, a number of implementation variants would be available. In the third lesson, you used the default implementation variant for a bag, which is a List implementation. Other implementation variants are:

- Bag as Table
- Bag as Diluted Table
- Bag as Hash Table

For this lesson, you will use the code from the third lesson as a starting point, and change the default Bag implementation.

Lesson 5: Changing the Implementation Variant

Setup

Copy the files `bike.h`, `bike.C`, `lesson.C`, and `main.C` from the `lesson3` directory (not the `lesson4` directory) to the `lesson5` directory, and then change your current directory to the `lesson5` directory. Use an editor to modify the files as described below.

Implementation

To change the default implementation of a collection to another implementation variant, you need to change the Collection Class file that you include, the collection typedef, and potentially the element and key functions.

Implementation Variant Header Files: To determine the correct header file to include, consult the “Class Implementation Variants” section of the chapter on Bag in the *OS/390 C/C++ IBM Open Class Library Reference*. The header file to include for `IBagAsDilutedTable` is shown as `ibagdil.h`. You therefore change the header file to include as follows:

```
// in lesson.C
// old:
/* #include <ibag.h> */
// new:
#include <ibagdil.h>
```

Templates for Implementation Variants: To determine the correct template to instantiate for the collection typedef, see the implementation variant in the appropriate collection chapter. In this case, you would look for “Bag as Diluted Table” in Chapter 16, “Bag” in the *OS/390 C/C++ IBM Open Class Library Reference*. The collection is shown there as:

```
IBagAsDilutedTable <Element>
IGBagAsDilutedTable <Element, COps>
```

Because you are not defining a generic operations class, you need to use the first implementation variant. You therefore change the typedef for the collection as follows:

```
// old: typedef IBag <Bicycle> MyCollectionType;
// new:
typedef IBagAsDilutedTable <Bicycle> MyCollectionType;
```

Element Type Functions: To determine the required element type functions, see the “Element Type” section for the implementation variant. In the case of `IBagAsDilutedTable`, the only element type function listed that was not listed for a Bag is the default constructor, which is already defined in `Bicycle` for other reasons. If other functions are required for a given implementation variant you choose to use in an application, use the information on implementing a hash function in Lesson 4 for hints on where to place and how to code such functions.

No further changes are required. For this lesson, the only implementation variant that would require additional element type functions is `IBagAsHshTable`, and the required additional function is a hash function, which is already described in “Lesson 4: Changing the Collection” on page 163.

Running the Program

The program should have the same behavior, for a given set of inputs, as the program from “Lesson 3: Changing the Element Type” on page 158. In a complex application, a change in performance might occur, but in all cases the behavior of a correctly coded program should be identical for different implementation variants of the same collection class.

What You Have Learned

Once a C++ program using the Collection Classes is functionally complete and error-free, you can focus on performance. The key to good performance of Collection Classes programs is to select the appropriate implementation variant of a given collection. Although this lesson did not explain which implementation variant to choose (since this is largely dependent on the class type being used in the collection and on other factors beyond the scope of the lessons), it showed you how to change the implementation variant once the appropriate variant has been selected. See “Features of Provided Implementation Variants” on page 123 for guidance on what implementation variants to select for a given application.

Errors When Compiling or Running the Lessons

If you code the programs in this chapter exactly as shown, they should compile successfully, and should run without any errors except those related to incorrect user input. Check your code for typographical mistakes or incorrectly placed code if you get compiler errors.

If you implement element, key, input, or output functions in different ways than those indicated, and your program does not compile successfully, or compiles but ends with an exception message when run, you can use Chapter 16, “Solving Problems in the Collection Class Library” on page 177 to determine the cause. You can also use Chapter 16 to find errors related to using a different collection or implementation variant from those specified in the lessons.

Other Tutorials

The Collection Class Library tutorials provided with OS/390 C++ compiler can help you to learn the concepts of the Collection Classes. They are presented in the same order as the Collection Class Library topics in this book. You should be familiar with the information in the first three chapters of Part 3 before beginning the tutorials.

Using the Default Classes

When you are learning to use a particular collection, you should first use the default class of that collection, so that you can gain a fundamental understanding of the collection before you approach the implementation variants of the collection.

You need to understand the topics covered in the following sections to successfully complete the tutorials:

Tutorial 1 Use of default implementations (“Instantiation and Object Definition” on page 89)

Tutorial 2 Adding, removing and replacing elements in a collection (“Adding, Removing, and Replacing Elements” on page 90)

Tutorial 3 Use of a cursor, locating and accessing elements, and the use of applicators (“Cursors” on page 93, “Using Cursors for Locating and Accessing Elements” on page 94, “Iterating over Collections” on page 96)

Tutorial 4 Use of exceptions (Chapter 14, “Exception Handling” on page 143)

After completing the above tutorials, you should be acquainted with the basic features of the Collection Class Library. For a more thorough understanding of the library, use the tutorials described below.

Advanced Use

If you want to understand more advanced uses of the classes, use tutorials 5 and 6. You need to understand the topics covered in the following sections to successfully complete the tutorials:

Tutorial 5 Exchanging implementation variants (Chapter 10, “Tailoring a Collection Implementation” on page 121)

Tutorial 6 Using abstract base classes to write polymorphic functions (Chapter 11, “Polymorphism and the Collections” on page 131)

Source Files for the Tutorials

Each tutorial includes seven files. There are four partitioned data sets for the tutorials as a whole. The following table shows the PDS names, file names (where ? corresponds to the number of the tutorial, from 1 to 6), and the purpose of the files in that data set:

PDS Name	Member Name	Purpose
CBC.SCLBTUT	EXAMPLE?	The C++ program file with sections missing. Fill in the gaps in this file, as well as the gaps in the .H file.
CBC.SCLBTUT	TUTJCL	The JCL to use to compile, link, and run the EXAMPLE? tutorials. Use this JCL after you have filled in the gaps in the program.
CBC.SCLBTUT	SXAMPLE?	The solution of EXAMPLE?, with the gaps filled in. Once you have tried out your own solution, check the corresponding SXAMPLE? file to see how closely your solution matches the intended one.
CBC.SCLBTUT.H	PERSON?	The .H file with sections missing. Fill in the gaps in this file, as well as the gaps in the related C++ program file EXAMPLE?.
CBC.SCLBTUT.H	SPERSON?	The solution of PERSON?, with the gaps filled in. Once you have tried out your own solution, check the corresponding SPERSON? file to see how closely your solution matches the intended one.
CBC.SCLBTUTD	TUTOR?	The instructions for how to complete each tutorial, and some questions on the tutorial.

PDS Name	Member Name	Purpose
CBC.SCLBTUTD	SOLUT?	A set of hints on how to perform the steps described in the related TUTOR? data set member, and the answers to the questions in that member.

The objective of the tutorials is to apply the information you have learned about the Collection Class Library by adding the missing parts for each file. You can compare your solutions to the solutions provided.

Chapter 16. Solving Problems in the Collection Class Library

This chapter helps you solve problems that you may encounter when you use the Collection Class Library. The following table provides a short summary of each problem, and directs you to a section containing hints for a solution.

Problem Area	Problem Effect	Page
Cursor Usage	Unexpected results when using cursors	177
Element Functions and Key-Type Functions	Error messages indicating a problem in <i>istdops.h</i>	178
Key Access Function - How to Return the Key (1)	Error messages indicating a problem in <i>istdops.h</i> : a local variable or compiler temporary is being used in a return expression	179
Key Access Function - How to Return the Key (2)	Unexpected results when adding an element to a unique key collection	180
Definition of Key-Type Functions	Link step returns error message CBC3013	180
Exception Tracing	Unexpected exception tracing output on standard error	181
Declaration of Template Arguments and Element Functions (1)	Compiler messages indicating that an element type or one of its required element functions is not declared	181
Declaration of Template Arguments and Element Functions (2)	Compilation errors from symbols being defined multiple times	181
Declaration of Template Arguments and Element Functions (3)	Link errors from symbols being defined multiple times	182
Default Constructor	Compiler error messages indicating a problem with constructors	182

Cursor Usage

Effect

You get unexpected results when using cursors. For example, the `elementAt()` function fails for the given cursor or returns an unexpected element.

Reason

You have used an undefined cursor. Cursors become undefined when an element is added to or removed from the collection.

Solution

Cursors that become undefined must be rebuilt with an appropriate operation (for example, `locate()`) before they are used again. Rebuilding is especially important for removing all elements with a given property from a collection. Elements cannot be removed by coding a cursor iteration. Use the `removeAll()` function that takes a predicate function as its argument.

For more information about cursors, see “Cursors” on page 93 and “Removing Elements” on page 91.

Element Functions and Key-Type Functions

Effect

When compiled, your program causes a compiler error indicating a problem in *istdops.h*. The following are examples of such errors:

Message if key is missing

```
CBC.SCLBH.H(ISTDOPS)(166:1) : (E) CBC3013:  
  "key" is undefined.  
CBC.SCLBH.H(ISTDOPS)(160:1) : (I) CBC3207:  
  The previous message applies to the definition of template  
  "IStdKeyOps<Parcel,ToyString>::key(const Parcel&) const".
```

Message if hash is missing

```
CBC.SCLBH.H(ISTDOPS)(152:1) : (E) CBC3070: .  
  Call does not match any argument list for "::hash".  
CBC.SCLBH.H(ISTDOPS)(146:1) : (I) CBC3207:  
  The previous message applies to the definition of template  
  "IStdHshOps<ToyString>::hash(const ToyString&,unsigned long) const".
```

Message if == is missing

```
CBC.SCLBH.H(ISTDOPS)(81:1) : (E) CBC3054:  
  The "==" operator is not allowed between "const ToyString" and  
  "const ToyString".  
CBC.SCLBH.H(ISTDOPS)(80:1) : (I) CBC3207:  
  The previous message applies to the definition of template  
  "equal(const ToyString&,const ToyString&)".
```

Message if < is missing

```
CBC.SCLBH.H(ISTDOPS)(105:1) : (E) CBC3054:  
  The "<" operator is not allowed between "const ToyString"  
  and "const ToString".  
CBC.SCLBH.H(ISTDOPS)(103:1) : (I) CBC3206:  
  The previous 2 messages apply to the definition of template  
  "compare(const ToyString&,const ToyString&)".
```

Reason

Compiler error messages indicating a problem in *istdops.h* are related to the element and key-type functions that you must define for your elements. These functions depend on the collection and implementation variant you are using. The compilation errors listed above occur when the `key()` function, the `hash()` function, `operator==`, or `operator<` are required for your elements, but are defined with the wrong interface or not defined at all. Whether arguments are defined as **const** is significant. Compiler messages do not always point directly to the incorrect function. For example, a compare function with non-**const** arguments results in the compilation error:

The "<" operator is not allowed between "const ..".

Solution

Verify which element and key-type functions are required for the implementation variant of the collection you are using. You can find this information for each collection in the section pertaining to the collection under the heading “Template Arguments and Required Functions.”

For more information about element and key-type functions, see Chapter 9, “Element Functions and Key-Type Functions” on page 101.

Note that the same problem may be produced if function declarations and definitions are not properly separated between header files and source files. This situation is described in detail in “Declaration of Template Arguments and Element Functions (1)” on page 181.

Key Access Function - How to Return the Key (1)

Effect

You get a compiler warning similar to:

Message if key is passed by value

```
CBC.SCLBH.H(ISTDOPS)(166:1) : (W) CBC3285:
    The address of a local variable or compiler temporary is being used
    in a return expression.
CBC.SCLBH.H(ISTDOPS)(160:1) : (I) CBC3207:
    The previous message applies to the definition of template
    "IStdKeyOps<Word,int>::key(const Word&) const".
```

Reason

Compiler error messages indicating a problem in *istdops.h* are related to the element and key-type functions that you must define for your elements. These functions depend on the collection and implementation variant you are using. Your global-name-space function `key()` returns the key by value instead of by reference. A temporary variable is created for the key within the operator-class function `key`. The operator class function `key` returns the key by reference. Returning a reference to a temporary variable causes unpredictable results.

The key function must return a reference and must also take a reference argument. If the key function calls other functions to access the key, it must call those functions with a reference to the object as an argument, and those functions must return a reference to the key.

Solution

Verify that the global name-space function `key` correctly returns a **key const&** instead of **key**.

For more information on element and key-type functions, see Chapter 9, “Element Functions and Key-Type Functions” on page 101.

Key Access Function - How to Return the Key (2)

Effect

You are adding an element into a unique key collection, such as a key set or a map, and you are sure that the collection does not yet contain an element with the same key. Nevertheless, you get unexpected results:

`KeyAlreadyExistsException`, or the element is not added and the cursor is positioned to a different element.

Reason

This problem has the same cause as the problem described in “Key Access Function - How to Return the Key (1)” on page 179. However, you did not get the warning message described above, because you compiled with a lower warning level.

Solution

This problem has the same solution as that described in “Key Access Function - How to Return the Key (1)” on page 179.

Definition of Key-Type Functions

Effect

You are using a collection class with a key, and you get an error message during the link step indicating a problem in *istdops.h*. The following are examples of such errors:

Message if key() function is undefined

```
CBC.SCLBH.H(ISTDOPS)(176:1) : (E) CBC3013:  
"key" function is undefined.
```

Reason

You are using a collection class that requires the element class to provide a key and you chose to use the method of using a global `key()` function. You are using collection class methods in a source file but the header file with the same name as the source file does not contain a declaration (prototype) of the global key function.

While compiling the source file, which uses methods of the collection class, the OS/390 C++ compiler has created or modified a temporary source file in the `tempinc` directory. During the link step, this source file is compiled to resolve references to template code. The error message you encounter refers to this compilation. The source file in the `tempinc` directory contains include directives for the collection class template code. It also contains include directives for a header file of the same name as the source file that uses the collection class methods. The template code in *istdops.h* requires that the global `key()` function be known at compilation time. The only file that is included at this time is the header file with the same name as your source file. The problem is that the source file is not included at this time, so a definition or declaration of the global `key()` function in this file is not recognized by the compiler.

Solution

You must declare the global `key()` function in the header file with the same name as the source file that uses the collection class methods. The definition of the global `key()` function should be in the source file. If you are not sure which header file is meant by the message, look in the source file found in the *tempinc* directory.

Exception Tracing

Effect

You get unexpected exception tracing output on standard error, even though the related exception causing the output is caught.

Reason

For each exception raised, the trace function `write()` of class `IOException::TraceFn` is called and writes information about the raised exception to standard error. This trace function `write()` is called whether the related exception is caught or not.

Solution

To suppress the trace output, provide your own `IOException::TraceFn::write()` tracing function by subclassing `IOException::TraceFn` and register the subclass with `setTraceFunction()`.

Declaration of Template Arguments and Element Functions (1)

Effect

You get compiler messages when processing templates indicating that an element type or one of its required element functions is not declared.

Reason

The element type or element function is defined locally to the source file that contains the template instantiation with the element type as its argument. For more information, see the section on template instantiation in the *OS/390 C/C++ Language Reference*.

Solution

Move the corresponding declarations to a separate header file and include the header file from the source file.

Declaration of Template Arguments and Element Functions (2)

Effect

You get compilation errors from symbols being defined multiple times.

Reason

The template instantiation needs to include the type declarations it received as arguments. Your header files containing type declarations used in template classes may automatically be included several times.

Solution

Protect your header files against multiple inclusion by using the following preprocessor macros at the beginning and end of your header files:

```
#ifndef _MYHEADER_H_
#define _MYHEADER_H_ 1

:
#endif
```

Where `_MYHEADER_H_` is a string, unique to each header file, representing the header file's name.

Declaration of Template Arguments and Element Functions (3)

Effect

You get link errors from symbols being defined multiple times.

Reason

The template instantiation needs to include the type declarations it received as arguments. Your header files containing type declarations used in template classes might automatically be included several times.

Solution

Verify that you did not define functions in the header files that declare types used in templates. If you did, you must move them from the header file into a separate source file or make them inline.

Default Constructor

Effect

You get a compiler error similar to the following:

Message for missing default constructor

```
CBC.SCLBH.H(ITBSEQ)(25:1) : (E) CBC3222:
"ITabularSequence<ToyString,IStdOps<ToyString> >::Node" needs a
constructor because class member "ivElement" needs a constructor
initializer.
Names namesOfExtinct(animals.numberOfDifferentKeys());
CBC.SCLBH.C(ANIMALS)(55:57) : (I) CBC3207:
The previous message applies to the definition of template
"ITabularSequence<ToyString>".
```


Reason

Compiler error messages indicating a problem with constructors for a collection are typically related to the constructors defined for your element. Here the default constructor for the element is missing.

Solution

Define the default constructor for the element class.

For more information about element and key-type functions, see Chapter 9, “Element Functions and Key-Type Functions” on page 101. The element and key-type functions required for each collection are listed for each collection type in sections entitled “Template Arguments and Required Functions.” Ensure that you used the prelinker before trying to link your text decks.

Chapter 17. Compatibility Information

This chapter tells you how the changes to the Collection Class Library can affect existing programs and how you develop, compile, and link future applications that use the library.

“Compatible Items” describes changes that do not affect compatibility with prior releases. These changes mainly affect the internal implementation structure of the library. However some of these changes also affect how you use the collections. For these changes, source code compatibility with former releases is maintained in nearly all cases.

“Incompatible Items” on page 186 identifies situations in which you need to recompile existing applications.

You should also read *OS/390 C/C++ SOM-Enabled Class Library User's Guide and Reference*. This book provides information on which libraries to use if you do not want to re-compile and re-link your existing applications when future releases of the Collection Class Library may become available.

Compatible Items

This section deals with items of former releases that are compatible with the new release.

Reference Classes

Within the new release reference classes are no longer necessary for polymorphic use of the collections. As shown in Figure 10 on page 84, the concrete collection classes are now directly derived from the abstract class hierarchy. A linkage of abstract and concrete classes through reference classes is therefore superfluous. Nevertheless you can continue using the reference class syntax in existing programs.

Iterator, IConstantIterator

The classes `Iterator` and `IConstantIterator` are now called `IApplicator` and `IConstantApplicator`. The new names express more precisely what the purpose of objects from these classes is: They do not iterate over a collection themselves but they provide a function that is applied to the elements of a collection during iteration with `allElementsDo()`.

The classes `Iterator` and `IConstantIterator` are still available but not recommended.

forCursor macro

Instead of the `forCursor` macro the `forICursor` macro is introduced. The `forCursor` macro is still available but - as with the `Iterator` classes - you should prefer using the new version.

IECops

Up to now all implementation variants of the collections bag, set, sorted bag and sorted set used the element operation class `IECops`. In the new release these collections require only class `ICops` which is a subset of `IECops`. That means, in the new release class `IECops` is no longer needed, yet it is still available.

Incompatible Items

Naming Conventions

New names have been introduced for the implementation variants as well as for the corresponding header files. The old names can still be used in existing programs. Consider the key set as example:

Old Names		New Names	
IKeySet	ikerset.h	IKeySet	iks.h
		IKeySetAsAvlTree	iksavl.h
IKeySetOnBSTKeySortedSet	iksbst.h	IKeySetAsBstTree	iksbst.h
IHashKeySet	ihshks.h	IKeySetAsHshTable	ikshsh.h
IKeySetOnSortedLinkedSequence	ikssts.h	IKeySetAsList	ikslst.h
IKeySetOnSortedTabularSequence	ikssts.h	IKeySetAsTable	ikstab.h
IKeySetOnSortedDilutedSequence	iksds.h	IKeySetAsDilTable	iksdil.h

Incompatible Items

This section lists items that are not compatible with the new collection class library release.

New class hierarchy

The structure of the Collection Classes changed in C/C++ for MVS/ESA Version 3 Release 1 Modification 1. All classes, including the concrete classes, are now related in an abstract hierarchy.

The abstract hierarchy makes use of virtual inheritance. When you subclass from a Collection Class and implement your own copy constructor, you must initialize the virtual base class `IACollection<Element>` in your derived classes. Therefore, if you subclassed from a concrete Collection Class that was shipped with C/C++ for MVS/ESA Version 3 Release 1 Modification 0, and are migrating to the Collection Classes that are shipped with OS/390 Release 3 C/C++, you will have to change the implementation of your copy constructor by adding the virtual base class initialization.

newCursor method

As opposed to former releases the return type of the `newCursor` method is now for any collection a pointer to the abstract cursor class `ICursor` (`ICursor*`).

Deriving from Reference Classes

Deriving from reference classes without overriding existing collection class member functions is still possible. Yet, you can no longer override existing collection class functions *and* use your derived collection class in a polymorphic way without additional effort. For further information, see Chapter 11, "Polymorphism and the Collections" on page 131.

Changed Implementation for Bag and Sorted Bag

The implementation of Bag and Sorted Bag has been changed in OS/390 C/C++ Release 3. In the previous releases, the Bag implementation was based on Key Set, and the Sorted Bag implementation was based on Key Sorted Set. Now they are based on Key Bag and Key Sorted Bag, respectively. This means that the implementation variants AVL Tree and B* tree are no longer available. For compatibility, these implementation

variants are mapped to the List implementation. The default implementation of Bag and Sorted Bag has changed from AVL Tree to the List implementation. The old implementations are still available in the C++ SOM (RRBC) library. For a description of the new implementations, see Figure 13 on page 123.

Part 4. Application Support Class Library

This part tells you how to use the Application Support classes.

Chapter 18. Application Support Class Library	191
Organization of Classes	191
IBase Class	194
IVBase Class	194
String and Buffer Classes	195
Thread Safety	195
MBCS and National Language Support	195
Chapter 19. String Classes	199
Introduction to the String Classes	199
What You Can Do with Strings	200
IStringTest Class	212
Chapter 20. Exception and Trace Classes	215
Introduction to the Exception Classes	215
Catching Exceptions Thrown by Class Library Functions	217
Throwing Your Own Exceptions Using the Exception Classes	218
Macros Used with the Exception Classes	219
Using the ITrace Class	222
Chapter 21. Date and Time Classes	225
IDate Class	225
ITime Class	227
ITimeStamp Class	229
Chapter 22. Controlling Threads and Protecting Data	231
Accessing the Current Thread	232
Starting a Thread	232
Protecting Data	234
Chapter 23. The IBM Open Class Notification Framework	235
Notifiers and Observers	235
Notification Protocol	236
IBM C++ Notification Class Hierarchy	237
Chapter 24. Using the Binary Coded Decimal Class	239
Header File and Constants for IBinaryCodedDecimal	239
Constructing IBinaryCodedDecimal Objects	240
IBinaryCodedDecimal Input and Output	240
Mathematical Operators for IBinaryCodedDecimal	240
Converting IBinaryCodedDecimal Objects	241
Number of Digits of an IBinaryCodedDecimal Object	242
Precision of an IBinaryCodedDecimal Object	242
IBinaryCodedDecimal Object Exceptions	242
Chapter 25. Using the Decimal Class	243
Header File	243

Constructing Decimal Objects	243
Decimal Class Input and Output	244
Operators for Decimal Class	244
Converting Decimal Objects	245
Number of Digits in a Decimal Object	246
Precision of a Decimal Object	246
Decimal Object Exceptions	247

Chapter 18. Application Support Class Library

The Application Support Class Library was developed by IBM, originally as part of the User Interface Class Library on C Set ++ for OS/2. Because these classes did not have the graphical-user-interface orientation of other classes in the User Interface Class Library, the classes were separated from the User Interface Class Library into a library of their own. On some operating systems, this class library is known as the "Data Types and Exceptions Class Library."

Organization of Classes

Figure 25 on page 192 shows the organization of the Application Support classes that are derived from IBase and those that are derived from IException. Five other classes do not inherit from any classes and are used to support the derived classes. See Table 7 on page 194 for information on the names of these classes and the classes they support. The purposes of the principal classes are described below. Classes are listed alphabetically.

IApplication	You can use this class to maintain a static pointer to the C++ object representing the currently executing application.
IBase	The base class of most of the other classes in the Application Support Class Library. This class provides an output operator and conversion functions for the library, and typedef synonyms used by other library classes to make programming easier. You do not need to create objects of the IBase class; it is described for completeness only.
IBaseErrorInfo	The IBaseErrorInfo class is an abstract base class that defines the interface for its derived classes. These classes retrieve error information and text that is then put into an exception object.
IBinaryCodedDecimal	The IBinaryCodedDecimal and decimal classes allow you to represent numerical quantities accurately in business and commercial applications for financial application.
IBuffer	Objects of the buffer classes contain the actual character contents of objects of the string classes. All manipulation of string characters is done in the buffer object referenced by the string object. IBuffer is the buffer class for single-byte character set objects.
IDate	This class provides support for date information. You can construct IDate objects in a number of ways, and then use IDate methods to determine the day of the week, month or year, compare two dates, test a date for certain characteristics, and obtain the names of days or months that are dependent on the national-language locale setting in effect at run time.
IDBCSBuffer	This class is the buffer class for multiple-byte character sets. Multiple-byte character sets are used for handling languages such as Japanese, Chinese, and Korean, which

Class Organization

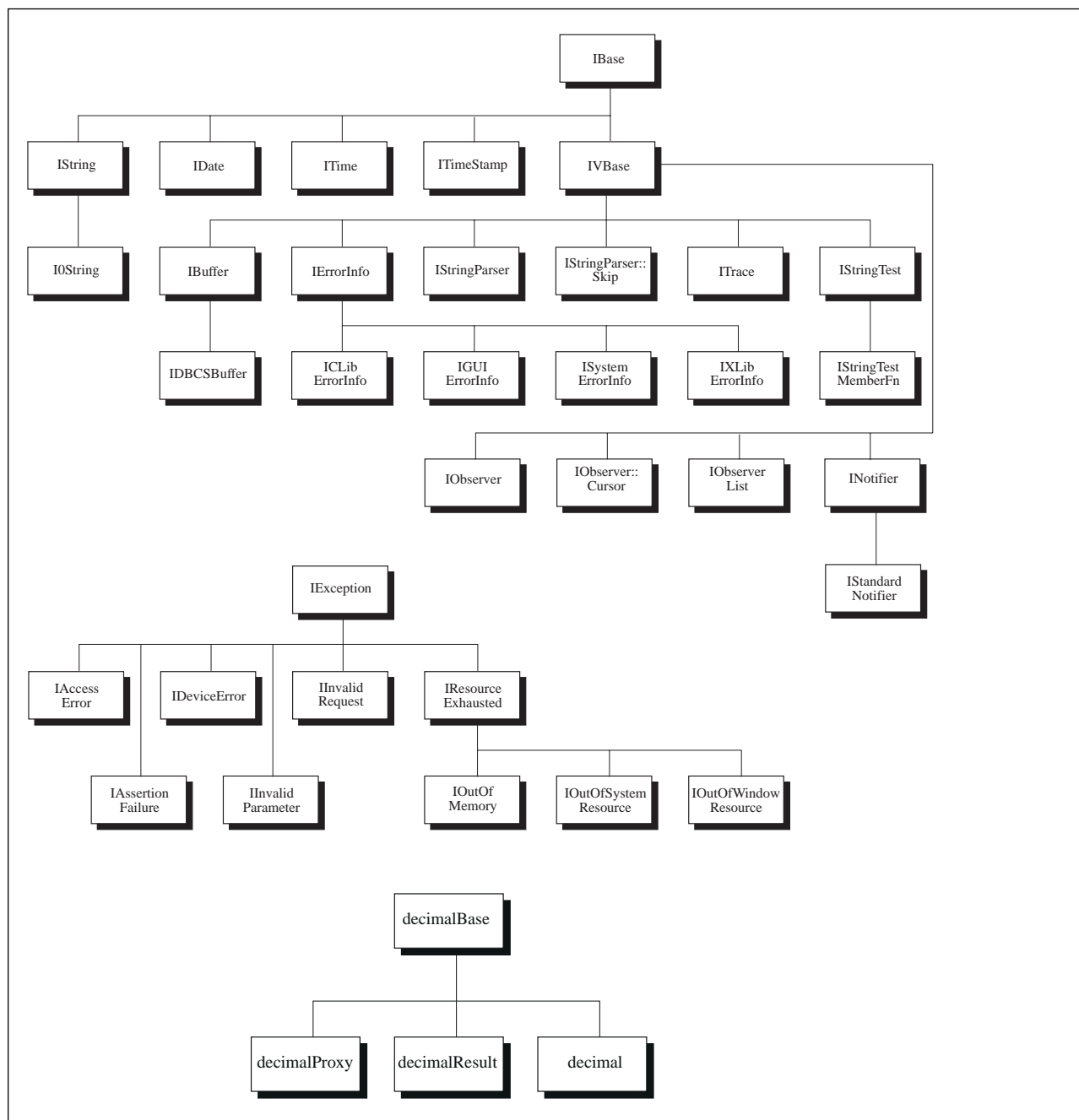


Figure 25. Organization of Application Support Class Library. Some class names have been split into two lines to fit in their boxes. Note that IGUIErrorInfo, IXLlibErrorInfo, and IOutOfSystemResource are not supported on OS/390 C/C++. The classes IDecimalUtil, decimalBase, decimalProxy, and decimalResult are meant for internal use by the Application Support Classes. Do not use them directly.

IException

contain more symbols than can be represented by the 256 characters of the single-byte character set.

The IException class is the base class from which all exception objects thrown in the library are derived.

IObserver

This class, along with the IObserverList, INotifier, IStandardNotifier, and IObserver::Cursor classes, lets you register observers with class objects so that you can be notified when a change to such an object takes place.

<code>IPrivateResource</code>	You can use this class to define a resource that is used within a single process.
<code>IRefCounted</code>	This class lets you maintain a count of all references to objects of the <code>IRefCounted</code> class.
<code>ISharedResource</code>	You can use this class to define a resource that can be shared across multiple processes.
<code>IString</code>	This class gives you a greater flexibility in handling strings than traditional C-style character arrays. The <code>IString</code> class supports both single- and multiple-byte character sets. With <code>IString</code> objects, you can code string-handling operations much more quickly. For example, you can concatenate two strings simply by using the <code>+</code> operator, or compare them using the <code>==</code> operator.
<code>IStringTest</code>	This class is provided so that you can define your own version of the matching function used by <code>IString</code> search and compare methods.
<code>IThread</code>	You can use this class to implement multithreaded applications.
<code>ITime</code>	You can use this class to create time-of-day objects and to compare them, add them together, extract specific information from them, or write them to an output stream.
<code>ITimeStamp</code>	You can use this class to create timestamp objects and to compare them, add them together, extract specific information from them, or write them to an output stream.
<code>ITrace</code>	Objects of the <code>ITrace</code> class provide module tracing. Whenever an exception is thrown by the library, trace records are output with information about the exception. You can use environment variables to redirect the trace output to a file.
<code>IVBase</code>	This class is a virtual base class used to derive other classes such as the buffer classes.
<code>I0String</code>	This class is identical to the <code>IString</code> class, except in its method of indexing strings. In the <code>IString</code> class, the first character of a string is at position 1, whereas the same string when stored in an <code>I0String</code> object has its first character at position 0. <code>I0String</code> is provided for programmers who are used to the C string-handling approach of treating strings as starting at position 0. <code>IString</code> and <code>I0String</code> objects are easily interchanged, and they support the same set of methods and operators.

One of the most important classes from a programmer's perspective is the `IString` class. This class can make your programming much more productive if you do any amount of string handling. The `IString` class provides a simpler, safer, and more flexible way of handling strings than traditional C-style character arrays and the functions of the `string.h` library. The `IString` class has associated classes that give you even greater flexibility in how you index strings and in how you test for pattern matches in the searching and replacing functions the class provides.

Table 7. Support Classes for Application Support Classes

Class Name	Supports These Classes
IStringEnum	IString IOString IBuffer IDBCSBuffer
IMessageText	IBase
IException::TraceFn	IException
IExceptionLocation	IException

IBase Class

The IBase class provides:

- An output operator for the library
- Conversion functions for the library
- Handling of the message text file
- Types for the library
- Synonyms

You do not need to create objects of the IBase class. This class is introduced at the root of the class hierarchy for the following reasons:

- To define the local type `Boolean` and the enumeration values `true` and `false`. This definition enables these identifiers to be referenced without their scope qualifier `IBase::` within declarations and member function definitions of classes derived from `IBase`.
- To provide basic functions applicable to many of the classes in IBM Open Class Library. These functions are `asString()`, `asDebugInfo()`, and `operator<<(ostream&)`. Note that `asString()` and `asDebugInfo()` do not work correctly if they are invoked through a pointer or reference to an `IBase` object, because the functions are not virtual. `IVBase` redeclares these as virtual functions. This means that, if you invoke these functions against an `IVBase*` or `IVBase&` object, the implementation for the actual class of the pointed-to or referenced object is invoked.

IVBase Class

The IVBase class:

- Ensures generic behavior for library classes that have virtual functions
- Allows derived classes to access the type and value names of the `IBase` class

All functions in the `IVBase` class should be overridden in derived classes because the `IVBase` class does not have access to any useful information about objects of its derived classes.

String and Buffer Classes

You can store and manage strings using the string and buffer classes. There are two type of string classes, two types of buffer classes, and two support classes. The two string classes, `IString` and `I0String`, are the main classes. The buffer and support classes are used to implement the string classes.

The buffer classes, `IBuffer` and `IDBCSBuffer`, contain the actual contents of the string objects. The `IDBCSBuffer` class supports characters of the multiple-byte character set (MBCS). The name of the class is `IDBCSBuffer` instead of `IMBCSBuffer` for compatibility reasons, because OS/2 implements a double-byte character set (DBCS) buffer, `IDBCSBuffer`. If you are using the string classes, MBCS support is automatic and transparent.

`IBuffer` and `IDBCSBuffer` are purely internal classes used in the implementations of `IString` and `I0String`. They are only used in protected sections of the `IString` class. They are described in this guide because you may want to understand them if you are deriving classes from `IString`.

The support classes, `IStringEnum` and `IStringTest`, provide data types and testing functions that are used in the string and buffer classes.

Thread Safety

The Application Support Class Library is thread safe at the class level. The class library protects the use of internal static and global data with locks. This means that while it is safe to have multiple threads manipulating different objects of the same class, it is not safe to have multiple threads manipulating the same object. If an object must be shared across threads, then you must protect access to it using appropriate serialization/coordination techniques.

For more information, see Chapter 22, “Controlling Threads and Protecting Data” on page 231.

Note: To run in a multi-threaded environment, the OS/390 UNIX kernel must be available and active.

MBCS and National Language Support

The library provides multiple-byte character set (MBCS) support and national language support (NLS). You can use one source file for your application code and provide MBCS and NLS support by using separate resource files for the languages you support. The benefits of this organization include the following:

- The application is easy to maintain, because a single version of the application is used. This reduces the cost of maintaining your code.
- The application is easy to upgrade because only the source code is upgraded and then linked to the separate language files for different languages. This reduces the time and cost of upgrading your code because different language versions can be generated at the same time.

Because message strings are defined in message catalogs, they can be translated easily to your local language without changes to the source code.

You should note the following when creating an MBCS-enabled application:

- String manipulation is MBCS-enabled. The string classes support mixed strings that contain both SBCS and MBCS characters. Use the string testing functions to determine if a character is single byte or multiple byte.
- The `IDBCSBuffer` class ensures that the search functions do not match the second or any subsequent bytes of an MBCS character and that the bytes of an MBCS character will not be split.

National Language Support

OS/390 C/C++ provides national language support using the XPG/4 programming model, and using the locale-sensitive functions of the C runtime library.

When you enable NLS, member functions of the `IString`, `IDate`, `ITime`, and `ITimeStamp` classes become locale sensitive, in both SBCS and MBCS environments. The classes provide the following capabilities:

IString	Character string handling in SBCS and MBCS environments
IDate	Date formatting and manipulation functions
ITime	Time formatting and manipulation functions
ITimeStamp	Date and time formatting and manipulation functions

While the interfaces of these classes do not change when you enable NLS, the underlying semantics change to reflect locale requirements. For example, the compare family of `IString` functions no longer perform bitwise comparisons, but instead perform comparisons based on the string collation sequence defined by the current locale.

Turning on Internationalization Semantics

To turn on the internationalization semantics, use the `ICLUI_I18N` environment variable:

```
GO step parameter ENVAR(ICLUI_I18N=ON)
```

To turn off the internationalization semantics, set `ICLUI_I18N=OFF`. The semantics are off by default.

You can also turn internationalization on or off from within your program, using the `IString` class. The `IString` class provides three member functions that allow you to programmatically turn internationalization on or off, and test for internationalization:

```
static void enableInternationalization( Boolean enable = true);  
static void disableInternationalization();  
static Boolean isInternationalized();
```

Aside from the three new `IString` functions, the interfaces of the `IString`, `IDate`, and `ITime` classes have remained the same.

Setting the Locale

To use national language support you must set the locale for your program, using the `setlocale` function:

```
setlocale(LC_ALL, "");
```

The `setlocale` function call should be the first call in `main()`, before any `lString` variables are defined. Your program should call `setlocale()` only once.

You can provide the locale information for `setlocale()` in the `LANG` environment variable. When your program runs, it then reads the locale information from the `LANG` environment variable. For example, to have your program use the Japanese locale, set the locale information for Japan before running the program:

```
GO step parameter ENVAR(LANG=Ja_Jp.IBM-932)
```

You can also set locales for specific categories of information. See the *OS/390 C/C++ Run-Time Library Reference* description of the `setlocale()` function for more information.

Note: Any references to locale that occur **before** the call to `setlocale()` will use the C locale by default. For more information on setting locales, see the section on locales in the *OS/390 C/C++ Programming Guide*.

Warning: In the XPG/4 model, the locales are process scoped. In a multi-threaded environment, you will get unpredictable results if another `setlocale()` call is made in a different thread.

Chapter 19. String Classes

The string classes define a data type for strings and provide member functions that let you perform a variety of data manipulation and management activities. They provide capabilities far beyond those available with standard C strings and the `string.h` library functions.

The string classes have the following capabilities:

- String buffers are handled automatically.
- Strings can contain both SBCS and MBCS characters.
- Strings can be indexed by character or by word.
- Strings can contain null characters. (There are no restrictions on the contents of a string object.)

Member functions of the string classes allow you to:

- Use strings in input and output
- Access information about strings
- Compare strings
- Test the characteristics of strings
- Search for characters or words within a string
- Manipulate and edit strings
- Convert strings to and from numeric types
- Format strings by adding or removing white space

Introduction to the String Classes

There are two string classes: `IString` and `I0String`. They are identical except for the method each uses to index its characters. The characters of an `IString` object are indexed beginning at 1. `I0String` characters are indexed beginning at 0. See "Indexing of Strings" on page 200 for more information on the indexing of the string classes. The string class you should use depends on which indexing scheme you prefer or find easier to implement.

Objects of `IString` and objects of `I0String` can be freely intermixed in a program. Objects of one class can be assigned objects of the other. Arguments that require an object of one will accept objects of the other. You will only notice a difference between an `IString` and an `I0String` when you are using functions that use or return a character index value.

In this chapter, only the `IString` class is presented. However, for every function of the `IString` class, there is a corresponding and identically named function of the `I0String` class. The `I0String` version of each function accepts the same arguments and has the same return type as the `IString` version, except that all parameters of type `IString` become `I0String`. Any other differences between the `IString` and `I0String` versions of the function are noted in the function descriptions in the *OS/390 C/C++ IBM Open Class Library Reference*.

String Buffers

When you create an object of a string class, the actual characters that make up the string are not stored in the string object. Instead, the characters are stored in an object of a buffer class.

The use of a buffer object is transparent to you when using the string classes. A correctly sized buffer is automatically created when you create a string object. The buffer is destroyed when a string object is destroyed. When you manipulate or edit a string, you are actually manipulating and editing the buffer object that contains the characters of the string.

Multiple-Byte Character Set Support

Objects of the `IString` class and the `I0String` class can contain a mixture of single-byte characters and multiple-byte characters. All member functions allow for the mixture. The searching functions will not match a single-byte character with the second or subsequent byte of a multiple-byte character. Functions that return substrings will never separate the bytes of a multiple-byte character.

Although the multiple-byte characters are supported, you must be careful not to alter the contents of a string in a way that would corrupt the data. For example, the statement:

```
IString[n]='x';
```

would be an error if the `n`th byte of the `IString` was part of a multiple-byte character.

Indexing of Strings

Objects of the string classes are arrays of characters. There are two types of indexes used with the arrays. The first is a character index: each character is numbered from left to right starting at the number 1 in the `IString` class and the number 0 in the `I0String` class. Therefore in the `IString` "The dog is brown," the letter "i" has an index value of 9. In the `I0String` "The dog is brown," the letter "i" has an index value of 8.

The second type of index is the word index. In the word index, each white-space-delimited word is numbered from left to right starting at the number 1. The word index is the same for `IString` objects and `I0String` objects. Therefore in the `IString` "The dog is brown," the word "is" has an index value of 3. In the `I0String` "The dog is brown," the word "is" also has an index value of 3.

The only difference between objects of the `IString` class and objects of the `I0String` class is the starting value for the character index.

What You Can Do with Strings

This section describes the wide range of string handling capabilities provided by the `IString` class. If you have a particular task you want to learn about from the list below, you can look that task up now and find references to appropriate `IString` functions. If you want an overview of all the capabilities of the `IString` class, read the entire section. The tasks are:

- Creating and copying strings
- Doing string input and output
- Concatenating strings
- Finding words or substrings within strings
- Replacing, inserting, and deleting substrings
- Determining string lengths and word counts
- Extending strings
- Converting between strings and numeric data
- Converting between strings and different base notations
- Testing the characteristics of strings
- Formatting strings

Many of the `IString` operators and functions are overloaded to support both `IStrings` and arrays of characters as return types and arguments. For example, the comparison operators (`==`, `>`, `<`, `>=`, `<=`, `!=`) all support either two `IString` operands or one `IString` and one array of characters operand. The array of characters operand can be on either side of the comparison operator. See the descriptions of individual member functions in the *OS/390 C/C++ IBM Open Class Library Reference* to determine what combinations of `IString` and array of characters are supported for a given function or operator.

Creating and Copying Strings

You can create `IStrings` using constructors, and you can copy `IStrings` using copy constructors, assignment operators, and substring functions.

IString Constructors

You can use `IString` constructors that construct null strings, that accept a numeric argument and convert it into a string of numeric characters, or that translate one or more characters into an `IString`. You can also create a single string out of up to three separate buffers, whose contents are concatenated into the created `IString` object. The following example shows some of the above ways of creating `IString` objects:

```
#include <istring.hpp>
#include <iostream.h>
void main() {
    IString Number1(123);    // --> Number1  ="123"
    IString Number2(123.12); // --> Number2  ="123.12"
    IString Character('a');  // --> Character ="a"
    IString String1("a");    // --> String1  ="a"
    IString String2("and");  // --> String2  ="and"
    IString String3("a\0d"); // --> String3  ="a"
}
```

Note that the last string (`String3`) is initialized with only the first byte of quoted text. The null character in the **char*** constructor argument is interpreted by the compiler as a terminating null. However, the `IString` class does support null bytes within strings. To construct `String3` as the example intended, you could write:

```
//...
IString String3("and");
String3[2]='\0';
```

If this string is later copied to another string, the null character and following characters are also copied:

```
IString String4=String3;
String4[2]='N';           // --> String4  ="aNd"
```

Copying IStrings

The IString assignment operator and copy constructor both copy one string to another string. One of the strings can be an array of characters, or both may be IString objects. The IString assignment operator and copy constructor offer the following advantages over the strcpy and strdup functions provided by the C string.h library:

- When an IString object is copied, a new copy of the string is not made. Instead, the two strings point to the same buffer location. The object is only copied if one of the strings is changed. This means that, for strings that are copied but where neither the source string nor the copy is subsequently changed, performance is improved by the amount of time it would have taken to make the new copy.
- The notation is simple and intuitive. To copy String1 into String2, you simply code String2=String1. With strings defined using the traditional **char*** method, such an assignment merely copies a pointer to the original string. With IString objects, the assignment copies each byte of the string into the new string.
- You do not have to determine the length of the source string and allocate sufficient storage to store it in the target string before the assignment. IString takes care of allocating the storage for you, whether the target string is being constructed within the assignment or has already been constructed. This reduces the risk of memory violations. In the following example, String2 is constructed and initialized, and then copied to (its original contents are overwritten), while String3 is copy-constructed to contain a copy of String1. Notice that String2's length is extended by the assignment operation.

```
IString String1="A longer string than String2";
IString String2="A short string";
IString String3=String1;    // initialized to String1
String2=String1;           // extended to fit String1
```

- The string being copied can contain null characters anywhere within it, and the entire string will be copied.
- If you accidentally create an array of characters without the terminating null, the strcpy function may continue copying past the storage allocated for the string. This can cause storage violations, or, at the least, it can corrupt the data in the target string. The length of IString objects is not determined by a terminating null, so storage violations and corrupt target strings are less likely.

Creating Substrings of Strings

You can use the subString function to return a new IString object containing a portion of another IString. This function lets you create an IString containing the leftmost characters, rightmost characters, or characters in the string's middle. The following example shows calls to subString that create substrings with leftmost, rightmost, or middle characters:

CLB3ASST

```
// Using the subString method of IString

#include <iostream.h>
#include <istring.hpp>
```

```

void main() {
    IString All("This is the entire string.");

    // Left -> subString(1, length)
    IString Left=All.subString(1,5);

    // Middle -> (startpos, length)
    IString Middle=All.subString(6,14);

    // Right -> (string length - (substring length - 1) )
    IString Right=All.subString(All.length()-6);

    cout << "<" << All << ">\n"
         << "<" << Left << ">\n"
         << "<" << Middle << ">\n"
         << "<" << Right << ">" << endl;
}

```

This program produces the following output:

```

<This is the entire string.>
<This >
<is the entire >
<string.>

```

Doing String Input and Output

The `IString` class overloads the input and output operators of the I/O Stream Class Library so that you can extract `IString` objects from streams and insert `IString` objects into them. The input operator reads characters from the input stream until a white-space character or EOF is encountered. The `IString` class also defines a member function to read a single line from an input stream. The following example shows uses of the input and output operators for `IString` and the `lineFrom` function:

CLB3ASIO

```

//Using the IString I/O operators and the lineFrom function

#include <istring.hpp>
#include <iostream.h>

void main() {
    IString Str1, Str2, Str3;
    Str1="Enter some text:";
    char test[80];

    // Write prompt
    cout << Str1;
    // Get input
    cin >> Str2;
    // This only reads in one word of text, so we should
    // check to see if this was the only word on the line:
    if (cin.peek()!='\n') {
        // there's more text on this line so ignore it
        cin.ignore(1000,'\n');
    }
    // Change prompt
    Str1.insert("more ",Str1.indexOf(" text:"));
    // Write prompt again
    cout << Str1;
    // Get line of input
    Str3=IString::lineFrom(cin,'\n');
    // Write output
    cout << "First word of first input: " << Str2 << '\n'
         << "Full text of second input: " << Str3 << endl;
}

```

Finding Words or Substrings

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

```
Enter some text:Here is my first string
Enter some more text:Here is my second string
First word of first input: Here
Full text of second input: Here is my second string
```

Note that, although null characters are allowed within an `IString` object, a null character in an input string is treated as the end of the input, and a null character in an `IString` being written to an output stream ends the output of that `IString`.

Concatenating Strings

The `IString` class defines an addition operator (+) to allow you to concatenate two words together. An addition assignment operator (+=) lets you assign the result of the concatenation to the left operand. The `copy()` member function lets you create an `IString` consisting of multiple copies of itself or of another string. The following example shows ways of concatenating text onto the start or end of an `IString`:

CLB3ACON

```
// Concatenating strings

#include <iostream.h>
#include <istring.hpp>

void main() {
    IString Str1="Let ";
    IString Str2="us ";
    IString Str3="concatenate ";
    IString Str4="repeatedly ";

    IString Str5=Str1+Str2; // Add Str1 and Str2 and store in Str5;
    Str5+=Str3;             // Add Str3 to Str5
    Str4.copy(3);           // Copy Str4 several times onto itself
    Str5+=Str4;             // Add Str4 to Str5
    cout << Str5 << endl;  // Write String 5
}
```

This program produces the following output:

```
Let us concatenate repeatedly repeatedly repeatedly
```

Finding Words or Substrings within Strings

A wide range of functions are available to let you find words, substrings, patterns, or individual characters within a string. You can even do wildcard searches: for example, you can search through a string to find a substring that begins with the letters "Ar" followed by one or more characters, followed by the letters "rk".

The following example shows a number of the searching functions available for `IString` objects. Comments describe the type of search operation being carried out.

CLB3ASRC

```
// Searching for substrings

#include <iostream.h>
#include <istring.hpp>
```

```

void main() {
    IString Str1="This string contains some sample text in English.";
    IString Str2=Str1.subString(27); // positions 27 and following:
                                   // "sample text in English."
    cout << "The string under consideration is:\n\n"
         << Str1 << "\n\n";

    // 1. Count the number of occurrences of a substring within the string

    cout << "The substring \"in\" occurs "
         << Str1.occurrencesOf("in")
         << " times in the string.\n";

    // 2. Find the first occurrence of a substring:
    //     (Note that the substring can be a char, char*, or IString value)

    cout << "The letter 'x' first occurs at position "
         << Str1.indexOf('x') << ".\n";

    // 3. Find the first occurrence of any letter of those specified:

    cout << "One of the letters q, r, or s first appears at position "
         << Str1.indexOfAnyOf("qrs") << ".\n";

    // 4. Find the first occurrence of any letter other than those specified:

    cout << "The first letter that is not in \"Think\" "
         << "appears at position "
         << Str1.indexOfAnyBut("Think") << ".\n";

    // 5. Find the index of a word

    cout << "The third word starts at position "
         << Str1.indexOfWord(3) << ".\n";

    // 6. Find a match to a phrase, and return the position of the
    //     first matching word

    cout << "The phrase \"" << Str2 << "\" starts at word number "
         << Str1.wordIndexOfPhrase(Str2) << " of the string.\n";

    // 7. Do a wildcard search to see if the string starts with "Th",
    //     contains "co", and ends with "sh."

    cout << "Does the string match the wildcard search string "
         << "\"Th*co*sh.*\"?\n";
    if (Str1.isLike("Th*co*sh.*")) cout << "Yes.";
    else cout << "No.";

    cout << endl;
}

```

This program produces the following output:

The string under consideration is:

This string contains some sample text in English.

The substring "in" occurs 3 times in the string.

The letter 'x' first occurs at position 36.

One of the letters q, r, or s first appears at position 4.

The first letter that is not in "Think" appears at position 4.

The third word starts at position 13.

The phrase "sample text in English." starts at word number 5 of the string.

Does the string match the wildcard search string "Th*co*sh.*"?

Yes.

Replacing, Inserting, and Deleting Substrings

The ability to manipulate the contents of an `IString` is one of the greatest advantages of the `IString` class over the traditional method of using `string.h` functions to manipulate arrays of characters. Consider, for example, a function that perform the following changes on a string. Issues that you need to address when using arrays of characters, but that are handled for you by the `IString` class, are shown in parentheses:

1. Replace all occurrences of Blue with Yellow (string must be expanded by two characters for each replacement, and text after the replacement must be shifted out).
2. Replace all occurrences of Orange with Pink (string must be shortened by two characters for each replacement).
3. Delete the sixth word of the string. (How are words delimited? By spaces? Carriage returns? Tab characters? What about multiple adjacent whitespace characters?)
4. Insert the word Dark as the fourth word or at the end of the string if the string has fewer than three words. (String must be extended. How are words delimited? Do you add a space before or after the word?).

You can easily handle the above requirements using `IString` member functions. The sample function `fixString()` below implements the requirements. Numbered comments correspond to the numbers of the requirements:

CLB3AREP

```
// Inserting, deleting and replacing substrings

#include <iostream.h>
#include <istring.hpp>

void fixString(IString&);

void main() {
    IString Str1="Light Blue and Green are nice colors. ";
    Str1+="But so are Red and Orange.";
    cout << Str1 << endl;
    fixString(Str1);
    cout << Str1 << endl;
}

void fixString(IString &myString) {
    myString.change("Blue", "Yellow"); // 1. Change Blue to Yellow
    myString.change("Orange", "Pink"); // 2. Change Orange to Pink
    myString.removeWords(6,1); // 3. Remove words, starting at word 6,
                                // for a total of 1 word.

    int Word4=myString.indexOfWord(4);
    if (Word4>0) // 4. Insert "Dark" as fourth word
        myString.insert("Dark ",Word4-1); // or at end of string if string
    else // has fewer than 4 words. The
        myString+=" Dark"; // insertion occurs 1 byte before
    // word 4 (otherwise it inserts
    // in the middle of word 4).
```

This program produces the following output:

```
Light Blue and Green are nice colors. But so are Red and Orange.
Light Yellow and Dark Green are colors. But so are Red and Pink.
```


Determining String Lengths and Word Counts

You can determine not only the length of a string, but the number of words within the string, or the length of a particular word in the string. The length of a string is not affected by any null characters you insert in the middle of the string. (The `strlen` function of `string.h` treats any null character in an array of characters as a terminating null.)

The following descriptions assume that `ThisString` contains the text “This string has five words.”

The `length` and `size` functions both return the length of an `IString`. For example, `ThisString.size()` returns the value 26, as does `ThisString.length()`.

To determine the number of words in a string, use the `numWords` member function. For example, `ThisString.numWords()` returns the value 5.

To determine the length of a particular word, use the `lengthOfWord` member function. For example, `ThisString.lengthOfWord(3)` returns the value 3.

Extending Strings

With arrays of characters, unless you allocate more storage than originally required for a string, you can only extend a string by allocating a new chunk of storage, moving the existing string into the new area, and extending it there.

`IString` objects are automatically extended for you whenever an `IString` operator or function requires the extension. This lets you spend more time coding useful function, and less time trying to track down the source of memory violations or data corruption. You can even use the subscript operator to assign a value to a position beyond the end of the string. The following example, by indexing past the end of `ShortString`, causes the string to be padded with blanks up to position 119, and the letter “a” is added at position 120:

```
IString ShortString="A short string";
ShortString[120]='a';
```

The `+` and `+=` operators, the assignment operator, and all member functions that change the contents of a string automatically allocate additional storage for the string if that storage is required. This can drastically reduce the amount of string-handling code you need to write.

Converting between Strings and Numeric Data

The `IString` class provides a number of `as...` functions that convert from `IString` objects to numeric types. You can also convert from numeric types to `IString` objects by using the versions of the `IString` constructor that take numeric values as arguments. The following example shows various `IString` functions that convert between strings and numbers:

CLB3ACV2

```
// Conversion between IString and numeric values

#include <iostream.h>
#include <istring.hpp>
```

```
void main() {
    IString NumStr=1.4512356919E1;    // Initialized with a float value
    int Integer=NumStr.asInt();        // Convert to integer value
    float Float=NumStr.asDouble();     // C++ conversion rules allow asDouble's
                                     // result to be converted to float
    double Double=NumStr.asDouble();  // Convert to double value
    NumStr=688;                       // Assign another integer value

    cout.precision(20);               // Set precision of cout stream
    cout << "Integer: " << Integer << "\nFloat:  " << Float
         << "\nDouble: " << Double << "\nString: " << NumStr << endl;
}
```

This program produces the following output:

```
Integer: 14
Float:   14.512356758117676
Double:  14.512356919
String:  688
```

You can also change the base notation of IString objects containing integer numbers, by using the d2... functions, which convert from decimal to binary, hexadecimal, or character representations. Conversion functions are described in the next section.

Converting between Strings and Different Base Notations

You can use the format conversion functions to change the way the data in a string is represented. These functions are overloaded so that each function has two versions. The nonstatic version replaces the value of the string with the converted value. The static version preserves the original string and returns a new string object containing the converted value. For example:

```
aString.c2b();                // Changes value of aString
IString binaryDigits = IString::c2b( aString );
                               // Preserves value of aString
```

The conversion functions check the format of the source string to make sure it is compatible with the source format implied by the function name. For example, if you use the b2d function to convert a string from binary to decimal, the function first checks that the string contains only the digits '0' and '1'. If it contains any characters other than those allowed by the source type, the format conversion functions always return 0.

The following example shows the use of the conversion functions. If you examine both the example and the output provided below, you can see how to use the functions.

CLB3ACV1

```
// IString conversion functions

#include <istring.hpp>
#include <iostream.h>
enum Bases {Bin, Dec, Hex, Char};
IString Base[4]={"binary", "decimal", "hex", "character"};
IString NumStr;

void Show(int From, int To, IString& Result) {
    cout << NumStr << " in " << Base[From] << " is "
         << Result << " in " << Base[To] << '.' << endl;
}
```

```

void main() {
    IString NewStr;
    NumStr="122";
    NewStr=IString::d2b(NumStr); Show(Dec,Bin,NewStr);
    NewStr=IString::d2x(NumStr); Show(Dec,Hex,NewStr);
    NewStr=IString::d2c(NumStr); Show(Dec,Char,NewStr);
    NumStr="Hat";
    NewStr=IString::c2b(NumStr); Show(Char,Bin,NewStr);
    NewStr=IString::c2d(NumStr); Show(Char,Dec,NewStr);
    NewStr=IString::c2x(NumStr); Show(Char,Hex,NewStr);
    NumStr="5F";
    NewStr=IString::x2b(NumStr); Show(Hex,Bin,NewStr);
    NewStr=IString::x2d(NumStr); Show(Hex,Dec,NewStr);
    NewStr=IString::x2c(NumStr); Show(Hex,Char,NewStr);
    NumStr="0110100001101001";
    NewStr=IString::b2d(NumStr); Show(Bin,Dec,NewStr);
    NewStr=IString::b2x(NumStr); Show(Bin,Hex,NewStr);
    NewStr=IString::b2c(NumStr); Show(Bin,Char,NewStr);
}

```

The output from this program resembles the following. Depending on the code page and character set (ASCII or EBCDIC) of the system you are running the program on, the values may vary.

```

122 in decimal is 01111010 in binary.
122 in decimal is 7A in hex.
122 in decimal is : in character.
Hat in character is 110010001000000110100011 in binary.
Hat in character is 13140387 in decimal.
Hat in character is C881A3 in hex.
5F in hex is 01011111 in binary.
5F in hex is 95 in decimal.
5F in hex is ~ in character.
0110100001101001 in binary is 26729 in decimal.
0110100001101001 in binary is 6869 in hex.
0110100001101001 in binary is ÇÑ in character.

```

Testing the Characteristics of Strings

The `IString` class lets you test your strings to determine characteristics such as the following:

- Whether they represent valid hexadecimal, decimal, or binary values
- Whether they contain only letters, letters and numbers, uppercase letters, lowercase letters, or punctuation characters
- Whether they contain all SBCS or DBCS characters

This list covers only a few of the testing functions provided by `IString`.

The testing functions return a value of type `Boolean` or `IBoolean`, indicating either `True` or `False` for the tested characteristic. For example, the function `isBinaryDigits()` returns `false` for the `IString` value "1101121101." All testing functions return a value of `false` for null `IString`.

The testing functions all have names beginning with `is...`, because they ask a question, such as "is the `IString` made up only of binary digits?" For a complete list of the testing functions, see the *OS/390 C/C++ IBM Open Class Library Reference*. The following example shows how you can use a subset of these functions:

CLB3ATST

```
// Evaluating strings using the IString is... methods

#include <istring.hpp>
#include <iostream.h>

void evaluate(IString& StringToTest) {
    if (StringToTest.isPrintable())
        cout << "Evaluating the string " << StringToTest << ":" << endl;
    else
        cout << "Evaluating an unprintable string:" << endl;
    if (StringToTest.isDigits())
        cout << "    Contains only digits 0-9." << endl;
    if (StringToTest.isAlphabetic())
        cout << "    Contains only alphabetic characters." << endl;
    if (StringToTest.isAlphanumeric())
        cout << "    Contains only alphabetic and numeric characters." << endl;
    if (StringToTest.isBinaryDigits())
        cout << "    Contains only zeros and ones." << endl;
    if (StringToTest.isHexDigits())
        cout << "    Contains only hex digits 0-9, a-f, A-F." << endl;
    if (StringToTest.isControl())
        cout << "    Contains only ASCII values 00-1F, 7F." << endl;
    if (StringToTest.isLowerCase())
        cout << "    Contains only lowercase letters a-z." << endl;
    if (StringToTest.isUpperCase())
        cout << "    Contains only uppercase letters a-z." << endl;
    if (StringToTest.isSBCS())
        cout << "    Contains only SBCS characters." << endl;
}

void main() {
    IString Str[6];
    Str[0]="12345";           // numeric, hexadecimal
    Str[1]="abcde";           // alphabetic, hexadecimal
    Str[2]="10101";           // numeric, binary
    Str[3]="abCde";           // alphabetic, hexadecimal
    Str[4]="xyz12";           // alphanumeric, lowercase
    Str[5]="\x04\x06\x11\x12"; // control, unprintable

    for (int i=1;i<6;i++) evaluate(Str[i]);
}
```

The output from this program resembles the following. Depending on the code page and character set (ASCII or EBCDIC) of the system you are running the program on, the results may vary.

```
Evaluating the string abcde:
    Contains only alphabetic characters.
    Contains only alphabetic and numeric characters.
    Contains only hex digits 0-9, a-f, A-F.
    Contains only lowercase letters a-z.
    Contains only SBCS characters.
Evaluating the string 10101:
    Contains only digits 0-9.
    Contains only alphabetic and numeric characters.
    Contains only zeros and ones.
    Contains only hex digits 0-9, a-f, A-F.
    Contains only SBCS characters.
Evaluating the string abCde:
    Contains only alphabetic characters.
    Contains only alphabetic and numeric characters.
    Contains only hex digits 0-9, a-f, A-F.
    Contains only SBCS characters.
Evaluating the string xyz12:
    Contains only alphabetic and numeric characters.
    Contains only SBCS characters.
Evaluating an unprintable string:
    Contains only ASCII values 00-1F, 7F.
    Contains only SBCS characters.
```

Formatting Strings

You can insert padding (white space) into strings so that each string in a group of strings has the same length. The `center`, `leftJustify`, and `rightJustify` functions all do this; their names indicate where they place the existing string relative to the added white space. You provide the final desired length of the string, and the function adds the correct amount of white space (or removes characters if the string is longer than the final length you specify). For example:

CLB3AST1

```
// Padding IStrings

#include <istring.hpp>
#include <iostream.h>

void main() {
    IString s1="Short", s2="Not so short",
           s3="Too long to fit in the desired field length";
    s1.rightJustify(20);
    s2.center(20);
    s3.leftJustify(20);
    cout << s1 << '\n' << s2 << '\n' << s3 << endl;
}
```

This program produces the following output:

```
          Short
    Not so short
Too long to fit in t
```

If a string is too wide, you can strip leading or trailing blanks using the `strip...` functions:

CLB3AST2

```
// Using the strip... functions of IString

#include <istring.hpp>
#include <iostream.h>

void main() {
    IString s1, s2, s3, Long="      Lots of space here      ";
    s1 = s2 = s3 = Long;
    s1.stripLeading();
    s2.stripTrailing();
    s3.strip();
    cout << ">" << Long << "<\n"
         << ">" << s1 << "<\n"
         << ">" << s2 << "<\n"
         << ">" << s3 << "<" << endl;
}
```

This program produces the following output:

```
>      Lots of space here      <
>Lots of space here      <
>      Lots of space here<
>Lots of space here<
```

You can also change the case of an IString to all uppercase or all lowercase:

CLB3AST3

```
// Changing the case of IStrings

#include <iostream.h>
#include <istring.hpp>

void main() {
    IString Upper="MANY of THESE are UPPERCASE CHARACTERS";
    IString Lower="Many of these ARE lowercase characters";
    Upper.change("MANY","NONE").lowerCase();
    Lower.change("Many","None").upperCase();
    cout << Upper << '\n' << Lower << endl;
}
```

This program produces the following output:

```
none of these are uppercase characters
NONE OF THESE ARE LOWERCASE CHARACTERS
```

Other IString Capabilities

This section has described only a portion of the functionality of the IString class. Many functions described here are overloaded to provide a wider range of functionality, and many of the functions of the IString class were not described here. See the *OS/390 C/C++ IBM Open Class Library Reference* for complete descriptions of all the public IString functions.

IStringTest Class

The IStringTest class lets you define the matching function used in the searching and testing functions of the string and buffer classes. When a search string is passed to a searching or testing function, the search string and the string object are compared on a character-by-character basis. The characters are considered to match if they are identical. The IStringTest class allows you to define when characters are considered to match.

For example, you can implement a string test that locates a given occurrence of a particular character in a string:

CLB3AIST

```
// Using the IStringTest class

#include <istring.hpp>
#include <iostream.h>

class Nth : public IStringTest {
    char key;           // Specifies the character to look for
    unsigned count;     // Specifies which occurrence to find
public:
    //
    // Construct an Nth object as follows:
    // 1. Create an IStringTest instance whose function type is user,
    //    with a null character to start;
    // 2. Initialize the count to n
    // 3. Initialize the key to c
    //
    Nth(char c, unsigned n)
    : IStringTest(user,0), count(n), key(c) { }
```

```

//
// test function: accepts an int (the character to look for)
// checks if the character matches the key
// if so, decrements count
// eventually, count will equal zero if enough matches are found,
// so "return !count" will return true (-1)
// otherwise, "return !count" will return a value other than -1

virtual Boolean test (int c) const
{
    if (c == key)          // if it matches,
        ((Nth*)this)->count--; // decrement count
    return !count;         // return complement of count
                           // will be true (-1) if count==0
}

};

void main() {
    IString text="this is a test string";
    cout << "The fourth appearance of the letter t in the string:\n"
        << text << '\n' << "is at position "
        << text.indexOf(Nth('t',4)) << endl;
}

```

This program produces the following output:

```

The fourth appearance of the letter t in the string:
this is a test string
is at position 17

```

A derived template class, `IStringTestMemberFn`, is provided to support the use of the `IStringTest` class with any function that accepts its objects as an argument.

A constructor for `IStringTest` accepts a pointer to a C function. The C function must accept an integer as an argument and return a Boolean. Such functions can be used anywhere an `IStringTest` can be used. Note that this is the type of the standard C library functions that check the type of C characters, for example, `isalpha()` and `isupper()`.

Chapter 20. Exception and Trace Classes

This chapter outlines some of the ways that you can use the exception and trace classes. The exception classes are a set of classes that allow you to catch exceptions based on their type. The trace class `ITrace` allows you to conveniently put trace statements in your programs.

Introduction to the Exception Classes

There are three primary ways to use the exception classes:

1. Certain functions in IBM class libraries throw exceptions that are objects of the exception classes. If you are familiar with the characteristics of the exception classes, you can take advantage of the exception classes to make your code that uses the IBM class libraries more robust.
2. You can both throw and catch objects of the exception classes in your own code. The exception classes provide a convenient way to package information about an exception.
3. You can derive your own classes from the exception classes.

Characteristics of the Exception Classes

The exception classes have the following characteristics:

- A stack of exception message text strings. These strings allow you to describe the exception in detail.
- An error ID that lets you uniquely identify what error caused the exception.
- A severity code that lets you determine whether the exception can be recovered from or not.
- Information about where the exception was thrown.

The exception classes' member functions allow you to:

- Add information about where the exception was thrown
- Add text to the description of the exception
- Get the error ID of the exception
- Determine if the exception is recoverable
- Log the exception data
- Set the error ID of the exception
- Set the severity of the exception
- Set a trace function

Derivation of the Exception Classes

The exception classes consist of a base class `IException` and a set of derived classes:

- `IAccessError`
- `IAssertionFailure`
- `IDeviceError`
- `IInvalidParameter`
- `IInvalidRequest`
- `IResourceExhausted`
- `IDecimalDataError`

Exception Classes

In addition, IResourceExhausted has the following derived classes:

- IOutOfMemory
- IOutOfSystemResource
- IOutOfWindowResource

Note: OS/390 C/C++ does not support the IOutOfWindowResource class. It is listed here because versions of the class library on other operating systems do support it.

Because all these classes are derived from the IException class, a single catch statement can catch all of the exceptions that are objects of the exception classes. The following catch statement, for example, will catch any exception that is an object of one of the exception classes:

```
catch(IException &ie){  
    // ...  
    // code for all exception class exceptions  
}
```

On the other hand, if you wanted to deal with each kind of exception separately, you could have catch statements that looked like this:

```
catch(IAccessError &ia){  
    // ...  
    // code for IAccessError exceptions  
}  
catch(IAssertionFailure &iaf){  
    // ...  
    // code for IAssertionFailure exceptions  
}  
// ...
```

Situations in Which the Exception Classes Are Used

The following table lists the exception classes and the situations in which they are typically thrown:

Exception Class	Thrown When ...
IAccessError	A logical error occurs, such as "resource not found"
IAssertionFailure	The expression in an IASSERT macro evaluates to false
IDeviceError	A hardware-related error occurs
IInvalidParameter	An invalid parameter is passed
IInvalidRequest	An object is in the wrong state for a function
IResourceExhausted	A resource is exhausted or currently unavailable
IOutOfMemory	Memory is exhausted
IDecimalDataError	The integral part of a IBinaryCodedDecimal object is truncated as the result of any mathematical operation.

Note: IDecimalDataError applies only to the IBinaryCodedDecimal class.

Catching Exceptions Thrown by Class Library Functions

Under certain circumstances, member functions of the Collection and Application Support Class Libraries will throw exceptions that are objects of the exception classes. You can take advantage of this fact to make your code that uses these classes more robust.

An Example of the Subscript Operator Throwing an Exception

The subscript operator of the `IString` class can throw exceptions that are objects of the exception classes. If you use the subscript operator on an `IString` object that is declared **const**, the operator will throw an `InvalidRequest` exception if the index is out of the bounds of the `IString` object.

In the following piece of code, an `IString` object is declared **const**, and then the subscript operator is used with an index beyond the size of the object.

CLB3ASUB

// Example that causes a subscript out of bounds exception

```
#include <iostream.h>
#include <iexcept.hpp>
#include <istring.hpp>

void main() {
    try {
        const IString ConstStr = "OFF";
        cout << ConstStr[4] << endl;
    }
    catch(IException &ie)
    {
        cout << "Type of exception is: " << ie.name() << endl;
        cout << "Location of exception is: "
            << ie.locationAtIndex(0)->fileName() << endl;
        if (ie.isRecoverable())
            cout << "Exception is recoverable" << endl;
        else
            cout << "Exception is unrecoverable" << endl;
    }
}
```

Because the index is beyond the size of the `IString` object, the subscript operator throws an exception. When this code is run, the following output is produced:

```
Type of exception is: IInvalidRequest
Location of exception is: istring5.C
Exception is recoverable
```

Member functions in the Collections and User Interface class libraries also throw exceptions that are objects of the exception classes. If you call such functions within try blocks followed by a catch statement for `IException` exceptions, you can:

- Make your code more robust by detecting and dealing with exceptions that occur in class library calls.
- Determine why exceptions are occurring by examining the information that is passed back in the exception class object.

Throwing Your Own Exceptions Using the Exception Classes

In addition to catching exception class exceptions that are thrown by class library functions, you can also throw them in your own code. Throwing exception class exceptions in your own code has the following advantages:

- The exception classes provide a convenient package for exception information.
- If you use one of the predefined exception classes or derive one of your own from `IException`, you can use the same catch statement to catch exceptions that are generated by both class library functions and your own functions.

Consider the following simple example. The `getFirstChar` function calls the `IASSERTSTATE` macro with a `get` call as an argument. If the `get` call fails, it returns zero and the `IASSERTSTATE` macro throws an `IInvalidRequest` exception.

CLB3AOPE

```
// Using the IASSERTSTATE macro

#include <iostream.h>
#include <fstream.h>
#include <iexcept.hpp>

void openFile(fstream& fs, char *filename){
    fs.open(filename, ios::in);
}

char getFirstChar(fstream& fs) {
    char c;
    IASSERTSTATE(fs.get(c));
    return c;
}

void main() {
    char c;
    char * filename = "source.dat";
    fstream fs;
    openFile(fs, filename);
    try {
        c = getFirstChar(fs);
        cout << "Here is first character: " << c << endl;
    }
    catch(IException &ie)
    {
        cout << "Type of exception is: " << ie.name() << endl;
        cout << "Location of exception is: "
            << ie.locationAtIndex(0)->fileName() << endl;
        if (ie.isRecoverable())
            cout << "Exception is recoverable" << endl;
        else
            cout << "Exception is unrecoverable" << endl;
    }
}
```

Suppose that this example is run, and the `source.dat` file is not available. The call to `open` in the `OpenFile` function will fail. When `getFirstChar` is called within the `try` block, an exception will be thrown by the `IASSERTSTATE` macro. This exception will be caught by the `catch` statement in `main`, and the output will look something like this:

```
Type of exception is: IInvalidRequest
Location of exception is: iopen.C
Exception is recoverable
```

Macros Used with the Exception Classes

The exception classes support a set of macros that allow you to manage the exception classes conveniently. You can use these macros to throw exceptions and to declare and define subclasses of `IException` or one of its subclasses.

ITHROW

Accepts as input an object of any `IException` subclass. It expands to add the location information to the instance, logs all instance data, and then throws the exception.

IRETHROW

Accepts as input a predefined instance of any subclass of `IException` that has been previously thrown and caught. Like the `ITHROW` macro, it also captures the location information, and logs all instance data before rethrowing the exception.

IASSERTSTATE

This macro accepts an expression to be tested as input. The expression is *asserted* to be true, meaning that you anticipate that it is true and are stating so to the compiler. If it evaluates to false, it invokes the `IExcept__assertState` function, which creates an `IInvalidRequest` exception. Location information is added to the exception, which is then logged and thrown.

IASSERTPARAM

This macro accepts an expression to be tested as input. The expression is asserted to be true. If it evaluates to false, it invokes the `IExcept__assertParameter` function, which creates an `IInvalidParameter` exception. Location information is added to the exception, which is then logged and thrown.

EXCLASSDECLARE

Creates a declaration for a subclass of `IException` or one of its subclasses.

EXCLASSIMPLEMENT

Creates a definition for a subclass of `IException` or one of its subclasses.

EXCEPTION_LOCATION

Expands to create an instance of the `IExceptionLocation` class.

INO_EXCEPTIONS_SUPPORT

Provided in support of compilers that lack exception handling implementation. If it is defined, the `ITHROW` macro ends the program after capturing the location information and logging it, instead of throwing an exception. This macro may not work correctly on all compilers.

ITHROWGUIERROR

This macro takes as its only argument the name of the GUI function that returned an error code. It calls the `IGUIError::throwGUIError` function, which creates an `IGUIErrorInfo` instance and uses it to create an `IAccessError` instance, adds location information, logs out the exception data, and throws the exception. The exception severity is set to recoverable. Only use this macro if the error information that is retrievable by the `IGUIErrorInfo` class is available.

Note: This macro and the `IGUIErrorInfo` class are not supported on OS/390 C/C++. They are described because versions of C Set ++ on other operating systems do support them.

ITHROWGUIERROR2

This macro takes three arguments:

- The name of the GUI function that returned an error code
- One of the values of the `IErrorInfo::ExceptionType` enumeration, which indicates the type of exception to be created
- One of the values of the `IException::Severity` enumeration, which indicates the severity of the exception

Only use this macro if the error information that is retrievable by the `IGUIErrorInfo` class is available.

Note: This macro and the `IGUIErrorInfo` class are not supported on OS/390 C/C++. They are described because versions of the exception classes on other operating systems do support them.

ITHROWSYSTEMERROR

This macro takes four arguments:

- The error ID returned from the system function
- The name of the system function that returned an error code
- One of the values of the `IErrorInfo::ExceptionType` enumeration, which indicates the type of exception to be created
- One of the values of the `IException::Severity` enumeration, which indicates the severity of the exception

Why Use the Macros?

You can manage exceptions that are objects of the exception classes directly. You can call member functions directly to create objects, and query and set their values. You can also explicitly derive your own classes from the existing exception classes. Often, however, it is more convenient to use the macros provided by the exception classes.

Consider the example that used the `IASSERTSTATE` macro:

```
// Using the IASSERTSTATE macro

#include <iostream.h>
#include <fstream.h>
#include <iexcept.hpp>

void openFile(fstream& fs, char *filename){
    fs.open(filename, ios::in);
}

char getFirstChar(fstream& fs) {
    char c;
    IASSERTSTATE(fs.get(c));
    return c;
}

void main() {
    char c;
    char * filename = "source.dat";
    fstream fs;
    openFile(fs, filename);
    try {
        c = getFirstChar(fs);
        cout << "Here is first character: " << c << endl;
    }
}
```

```

catch(IException &ie)
{
    cout << "Type of exception is: " << ie.name() << endl;
    cout << "Location of exception is: "
        << ie.locationAtIndex(0)->fileName() << endl;
    if (ie.isRecoverable())
        cout << "Exception is recoverable" << endl;
    else
        cout << "Exception is unrecoverable" << endl;
}
}

```

This code could be rewritten to invoke the exception class member functions directly:

CLB3AMAC

```

// Invoking the IException member functions directly

#include <iostream.h>
#include <fstream.h>
#include <iexcept.hpp>

void openFile(fstream& fs, char *filename){
    fs.open(filename, ios::in);
}

char getFirstChar(fstream& fs) {
    char c;
    if (!fs.get(c)) {
        IInvalidRequest ir(" ", 0, IException::recoverable);
        IExceptionLocation il("imac.C","getFirstChar",5);
        ir.addLocation(il);
        throw(ir);
    }
    return c;
}

void main() {
    char c;
    char * filename = "source.dat";
    fstream fs;
    try {
        c = getFirstChar(fs);
        cout << "Here is first character: " << c << endl;
    }
    catch(IException &ie)
    {
        cout << "Type of exception is: " << ie.name() << endl;
        cout << "Location of exception is: "
            << ie.locationAtIndex(0)->fileName() << endl;
        if (ie.isRecoverable())
            cout << "Exception is recoverable" << endl;
        else
            cout << "Exception is unrecoverable" << endl;
    }
}

```

Notice how the single `IASSERTSTATE` in the `getFirstChar` function is replaced with a test of the return value of `get`, the definition of an `IInvalidRequest` object, the definition of an `IExceptionLocation` object, and an explicit `throw` statement. You can see that the version of the program that uses the `IASSERTSTATE` macro is simpler and easier to code.

Using the ITrace Class

The ITrace class provides a set of facilities that allow you to put trace statements in your code conveniently. The most convenient way to use ITrace is through the macros that it supports.

Using the Trace Macros to Control Trace Output

The ITrace class is convenient to use because it allows you to turn trace statements on and off easily. By defining certain macros and by using the macros in the ITrace class to create trace output, you can selectively turn tracing on and off. There are three special trace macros:

- IC_TRACE_RUNTIME
- IC_TRACE_DEVELOP
- IC_TRACE_ALL

By defining or not defining these macros, you can specify whether or not the trace macros are expanded, and thus whether or not your program produces trace output.

If IC_TRACE_RUNTIME is defined, the following macros are expanded:

IMODTRACE_RUNTIME

This macro takes one argument that is the name of the current module. It creates an ITrace object using the module name as the name of the trace and the current line number as the line number.

IFUNCTRACE_RUNTIME

This macro takes no arguments. It creates an ITrace object using the function name as the name of the trace and the current line number as the line number.

Note: On OS/390 the function name is always the value `f`, because the OS/390 C++ compiler does not support the `__FUNCTION__` macro.

ITRACE_RUNTIME

This macro takes a single argument. This argument is written to the trace location.

If IC_TRACE_DEVELOP is defined, all of the macros that are expanded when IC_TRACE_RUNTIME is defined, are also expanded. In addition, the following macros are expanded:

IMODTRACE_DEVELOP

This macro takes one argument. Typically you use the argument to name the current module. This macro creates an ITrace object using the module name as the name of the trace and the current line number as the line number.

IFUNCTRACE_DEVELOP

This macro takes no arguments. It creates an ITrace object using the function name as the name of the trace and the current line number as the line number.

Note: On OS/390 the function name is always the value `f`, because the OS/390 C++ compiler does not support the `__FUNCTION__` macro.

ITRACE_DEVELOP

This macro takes a single argument. This argument is written to the trace location.

If IC_TRACE_ALL is defined, all of the trace macros are expanded.

Capturing Trace Output in a File

The ITrace class allows you to send trace output to standard output, or to capture it in a file. To capture trace output in a file, you must define the following environment variables before starting your application:

```
ICLUI_TRACETO=FILE
ICLUI_TRACEFILE=<file>
```

where <file> is the name of the target output file.

An Example of Using ITrace

The following piece of code shows one way that you could use the trace macros to produce trace output for your programs. In this code, the macros IFUNCTRACE_DEVELOP and ITRACE_DEVELOP are used to create trace statements that indicate that the flow of control has passed through the functions openFile and getFirstChar.

CLB3ATRC

```
// Producing trace output with the ITrace class

#define IC_TRACE_DEVELOP

#include <iostream.h>
#include <fstream.h>
#include <iexcept.hpp>
#include <itrace.hpp>

void openFile(fstream& fs, char *filename){
    IMODTRACE_DEVELOP("openFile(fstream&,char*)");
    fs.open(filename, ios::in);
    ITRACE_DEVELOP("after open statement");
}

char getFirstChar(fstream& fs) {
    char c;
    IMODTRACE_DEVELOP("getFirstChar(fstream&)");
    fs.get(c);
    ITRACE_DEVELOP("after get statement");
    return c;
}

void main() {
    char c;
    char * filename = "source.dat";
    fstream fs;
    //
    // static functions to enable tracing and direct
    // tracing output to standard output
    //
    ITrace::enableTrace();
    ITrace::writeToStandardOutput();
    openFile(fs, filename);
    c = getFirstChar(fs);
    cout << "Here is first character: " << c << endl;
}
```

Notice that, in this code, the static functions `enableTrace` and `writeToStandardOutput` are used to enable tracing and to direct the trace output to standard output.

Because the macro `IC_TRACE_DEVELOP` is defined, the trace macros produce trace output. In addition, the trace output has been explicitly directed to standard output, so the output of the code looks like this:

```
+openFile(fstream&,char*)
  >after open statement
-openFile(fstream&,char*)
+getFirstChar(fstream&)
  >after get statement
-getFirstChar(fstream&)
Here is first character: t
```

Suppose that you wanted to turn off the trace output in this program. One way to do it is to modify the code so that the macro `IC_TRACE_DEVELOP` is not defined. If you do this, the trace macros are not expanded, and no trace output is produced. The output of this code with `IC_TRACE_DEVELOP` not defined looks like this:

```
Here is first character: t
```

Chapter 21. Date and Time Classes

The `IDate` and `ITime` classes are independent classes that provide you with data types to store and manipulate date and time information. Because the `IDate` and `ITime` classes are independent, when an `ITime` object's time passes 23:59:59 (24-hour format) or 11:59:59 (12-hour format), it has no effect on the value of any `IDate` object.

The `ITimestamp` class provides you with a data type to store and manipulate timestamp information, where a timestamp represents a specific point in time; for example, combined date and time.

With these classes, you can create date, time, and timestamp objects, and use member functions to do the following:

- Write date, time, or timestamp objects to an output stream
- Access detailed information about dates, times, or timestamps
- Compare dates, times, or timestamps
- Test the characteristics of date or time objects
- Add or subtract days from a date object
- Add or subtract hours, minutes, or seconds from a time or timestamp object
- Convert between date formats or between time formats.

IDate Class

The `IDate` class uses Gregorian calendar dates. The Gregorian calendar is in general use and consists of the 12 months, January to December.

`IDate` also supports the Julian date format, which contains the year in positions 1 and 2, and the day of the year in positions 3 through 5. If the day of the year is less than three digits, zeros are added on the left to increase the size to three digits. For example, February 14, 1965 is 65045 as a Julian date. (February 14 is the 45th day of the year.)

The `IDate` class returns the names of the days and months in the language defined by the current `locale`. For information on defining the `locale`, see the standard C library function `setlocale()`.

Creating an IDate Object

You can create an `IDate` object using different `IDate` constructors. For example:

```
IDate OneDay(IDate::June,30,1994);    // Month, day, year
IDate AnotherDay(23,IDate::April,1961); // Day, month, year
IDate SomeDay(940616);                // Julian date format
IDate Yesterday(1994,177);            // Year, day of year
```

The constructors accepting a month use the `IDate` enumeration `Month`, whose members are named January through December (the months of the year in English).

Changing an IDate Object

You can add days to, or subtract days from, an IDate object. You can also subtract one date from another, in which case the result is the number of days between the two dates. For example:

```
IDate Day1, Day2;
int NumDays;
Day1=IDate::today();
Day2=Day1+1;      // Day2 is one day after Day1
Day2+=2;          // Day2 is now three days after Day1
NumDays=Day2-Day1; // NumDays=3
```

Note that you cannot add two IDate objects together, because such an addition does not make sense. However, you can add two ITime objects together.

Information Functions for IDate Objects

The IDate class defines information functions that you can use to obtain specifics about an IDate object. For example, you can find out what day of the week, month, or year an IDate object's date falls on, or what the name of the day or month is for the current locale. You can also find out what today's date is. The following example shows some of the IDate information functions:

CLB3ADTF

```
// Information functions for IDate class

#include <iostream.h>
#include <istring.hpp>
#include <idate.hpp>

void main () {
    IDate Day1(27, IDate::May, 1964);
    cout << Day1.dayName() << " "
         << Day1.monthName() << " "
         << Day1.dayOfMonth() << " out of "
         << IDate::daysInMonth(Day1.monthOfYear(), Day1.year()) << " days in month, "
         << IDate::daysInYear(Day1.year()) << " days in year "
         << Day1.year() << "." << endl;
}
```

This program produces the following output:

```
Wednesday May 27 out of 31 days in month, 366 days in year 1964.
```

Testing and Comparing IDate Objects

You can compare two IDate objects to determine whether they are equal, or whether one is later than the other. The following operators are defined: ==, !=, <, <=, >, >=. For example, the expression if ((Day1>Day2) && (Day1!=Day3)) evaluates to true if Day1 is January 1 1994, Day2 is June 3 1968, and Day3 is July 12 1941.

You can also check whether a particular year is a leap year, or whether a particular combination of day, month, and year is valid. The isLeapYear() function returns true if its integer argument is a leap year. The isValid() function accepts combinations of day, month, and year (or day of year and year), and returns true if the provided date is valid. For example, it returns true for the first date below, and false for the second date:

```
if (IDate::isValid(IDate::June, 30, 1990)) // ...
if (IDate::isValid(1965, 366))             // ... False (No day number 366 in 1965)
```

ITime Class

The ITime class refers to time in the 24-hour format by specifying time units (hours, minutes, seconds) past midnight. If you want to display ITime objects in the 12-hour format, you must convert them to IStrings using the asString function with a char* argument of "%r". (This argument is a format string. All format specifiers of the strftime() function of the standard C library are supported by the IString conversion function.)

Note: Objects of the ITime class are precise only up to the nearest second, and cannot be used for more precise timings.

Creating an ITime Object

You can create an ITime object and initialize it to a number of seconds past or before midnight, or to a number of hours, minutes, and optionally seconds past midnight:

```
ITime Time1(33556),      // 09:19:16
    // 33556 = 9 hours (32400 seconds), 19 minutes (1140 seconds),
    // 16 seconds (adds up to 33556)
Time2(-33556),          // 14:40:44
    // (9 hours, 19 minutes and 16 seconds BEFORE midnight)
Time3(12,00),           // 12:00:00 (noon)
Time4(3,3,3);           // 03:03:03
```

The constructors translate incorrect times into valid ITime objects using modulo arithmetic. For the seconds past midnight format, any number whose absolute value is greater than or equal to 86400 is divided by 86400, and the remainder is used to calculate the time. For the hours, minutes, and optional seconds format, excess minutes and seconds are added to the hours and minutes values, respectively, and if the hour exceeds 23 it is divided by 24 and the remainder is taken. For example:

```
ITime Time1(133556),    // 13:05:56 (13356-86400=47156 seconds after midnight)
Time2(-133556),         // 10:54:04 (13356-86400=47156 seconds BEFORE midnight)
Time3(10,119,60),       // 12:00:00 (noon) (10 hours plus 119 minutes plus 60 seconds)
Time4(33,33);           // 09:33:00 (33 hours - 24 hours = 9 hours)
```

Changing an ITime Object

You can add or subtract two times. Four operators are provided: +, +=, -, and -=. The following example shows the use of these operators:

```
ITime Start(12:00), Duration(2:00),
    Done=Start+Duration; // Done=14:00
Start=Done-Duration;     // Start=12:00 still
Start+=Duration;         // Start=14:00
Start-=Duration;         // Start=12:00 again
```

Information Functions for ITime Objects

Three of the information functions return an &time2's hour, minute, or second settings; the other information function returns the current time as determined by the system clock. For example:

```
ITime Time1(ITime::now());
cout << Time1.hours() << " o'clock occurred "
    << Time1.minutes() << " minutes and "
    << Time1.seconds() << " seconds ago." << endl;
```

This displays a result such as the following:

```
12 o'clock occurred 16 minutes and 23 seconds ago.
```

Comparing ITime Objects

Functions are defined to let you compare ITime objects for equality, inequality, or relative position in time. The following operators are defined: ==, !=, <, <=, >, >=. In the following example, a message is displayed if enough time elapses between the first and second calls to the now() member function:

```
#include <itime.hpp>
#include <iostream.h>
ITime First(ITime::now());
void main() {
    ITime Second=ITime::now();
    if (First<Second) // Some time has passed
        cout << "You must be debugging me!" << endl;
}
```

This message usually does not print when the program is run outside of a debugging session. However, if you debug the program and step through each line slowly, the message may be displayed, because the first ITime object is initialized during program initialization (before **main** is called) while the second ITime object is initialized within **main**.

Writing an ITime Object to an Output Stream

ITime defines an output operator that writes an ITime object to an output stream in the format hh:mm:ss. If you want to write the object out in a different format, you should convert the object to an IString using the asString member function. This member function accepts a char* argument containing a format specifier. The format specifier is the same one as used by the C library function strftime. The following program displays some valid specifiers and the output they produce:

CLB3ATIM

```
// Examples of ITime output

#include <istring.hpp>
#include <itime.hpp>
#include <iostream.h>
#include <iomanip.h> // needed for setw(), to set output stream width

void main() {
    char* FormatStrings[]={
        "%H : %M and %S seconds", // %H, %M, %S - 2 digits for hrs/mins/secs
        "%r",                     // %r - standard 12-hour clock with am/pm
        "%T",                     // %T - standard 24 hour clock
        "%T %Z",                  // %Z - local time zone code
        "%IM past %I %p"          // %I... - One digit for hour/minute
    };
    // %p - am/pm

    cout.setf(ios::left,ios::adjustfield); // Left-justify output

    cout << setw(30) << "Format String" // Title text
         << setw(40) << "Formatted ITime object" << endl;

    for (int i=0;i<5;i++) { // Show each time
        IString Formatted=ITime::now().asString(FormatStrings[i]);
        cout << setw(30) << FormatStrings[i]
             << setw(40) << Formatted << endl;
    }
}
```

Note: The format specifier %n, where n is an integer, is not supported by strftime on OS/390. As a result, if you use a format specification string containing %n in ITime output, the format specification string may appear in place of the desired output.

The program produces output that looks like the following:

Format String	Formatted ITime object
%H : %M and %S seconds	16 : 13 and 04 seconds
%r	04:13:04 PM
%T	16:13:04
%T %Z	16:13:04 EST
%IM past %I %p	13 past 4 PM

ITimeStamp Class

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. If the timestamp is referring to a point in time before the reference date, a negative value must be used.

Creating an ITimeStamp Object

You can create an ITimeStamp object using different ITimeStamp constructors. For example:

```

IDate ADate(IDate::December, 5, 1963);           // Create an IDate object
ITime ATime(10, 11, 12);                         // Create an ITime object

ITimeStamp TmStamp1(ADate);                      // 12/05/1963 midnight
ITimeStamp TmStamp2(ADate, ATime);               // 12/05/1963 10:11:12 am
ITimeStamp TmStamp3(4000.0);                     // 01/01/2000 01:06:40 am
ITimeStamp TmStamp4(-4000.0);                    // 12/31/1999 22:53:20 pm
ITimeStamp TmStamp5;                             // same as ITimeStamp TmStamp5(0.0);
                                                // 01/01/2000 00:00:00 am

```

Changing an ITimeStamp Object

You can add seconds to, or subtract seconds from, an ITimeStamp object. You can also subtract one ITimeStamp object from another, in which case the result is the number of seconds between the two timestamps. For example:

```

ITimeStamp TmStamp1, TmStamp2;
double diff;
TmStamp1 = ITimeStamp::currentTimeStamp();
TmStamp2 = TmStamp1 + 4000.0;                     // 4000.0 seconds after TmStamp1
TmStamp2 -= 1000.0;                              // go back 1000.0 seconds
diff = TmStamp2 - TmStamp1;                      // should be 3000.0 seconds different
                                                // (if there is no rounding error)

```

Note: You cannot add two ITimeStamp objects together, as such an addition does not make sense. Also, all the operations are done using floating point arithmetic. As a result, some error due to rounding may occur.

Information Functions for ITimeStamp Objects

The ITimeStamp class defines information functions that you can use to obtain specific information about an ITimeStamp object. For example, you can determine the number of seconds separating the ITimeStamp object from the reference date (01/01/2000 00:00:00). You can also find out what the current timestamp is.

Conversion operators have been provided that allow you to convert an existing ITimeStamp object to an IDate object or an ITime object. Once the object has been converted, the IDate or ITime information functions may be then be used. See “Information Functions for IDate Objects” on page 226 and “Information Functions for ITime Objects” on page 227 for more information.

The following example shows some of the ITimeStamp information functions:

Comparing ITimeStamp Objects

```
ITimestamp RefDate;
ITimestamp TmStamp = ITimeStamp::currentTimeStamp();

IDate ADate = TmStamp;
ITime ATime = TmStamp;

double Seconds = TmStamp.asSeconds();

cout << TmStamp << " is " << Seconds << " seconds apart from" << endl;
cout << RefDate << endl;
cout << ATime.hours() << ":" << ATime.minutes() << ":";
cout << ATime.seconds() << "," << ADate.dayOfYear();
cout << " days in year " << ADate.year() << endl;
```

This example produces the following output:

```
05/15/1996 17:50:56 is -1.14502e+08 seconds apart from
01/01/2000 00:00:00
17:50:56, 136 days in year 1996
```

Comparing ITimeStamp Objects

You can compare two ITimeStamp objects to determine whether they are equal, or whether one is later than the other. The following operators are defined: ==, !=, <, <=, >, and >=.

Note: Since all the operations are done using floating point arithmetic, be aware that some rounding error may occur.

The following example illustrates this point:

```
ITimestamp TmStamp1(12345.54321);
ITimestamp TmStamp2 = TmStamp1 + 9753.6802 - 9753.6802;

if (TmStamp1 == TmStamp2)
{
    printf("TmStamp1 == TmStamp2\n");
    printf("TmStamp1 = %30.20f\n", TmStamp1.asSeconds());
    printf("TmStamp2 = %30.20f\n", TmStamp2.asSeconds());
}
else
{
    printf("TmStamp1 != TmStamp2\n");
    printf("TmStamp1 = %30.20f\n", TmStamp1.asSeconds());
    printf("TmStamp2 = %30.20f\n", TmStamp2.asSeconds());
}
```

This examples displays the following output:

```
TmStamp1 != TmStamp2
TmStamp1 =      12345.5432100000000000000000
TmStamp2 =      12345.54320999999800000000
```

Chapter 22. Controlling Threads and Protecting Data

The Application Support Class Library provides classes to implement multithreaded applications. This means that your application can run multiple threads concurrently, and each thread will execute regardless of whether the other threads give up control. A thread is defined to be the smallest unit of execution within a process which maintains the processor state and program stack.

The primary class you use to handle threads is `IThread`. Objects of this class represent separate threads of execution and provide the ability to start and stop the thread, set various thread attributes, and determine the default environment for the thread.

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`.
- To create additional threads of execution by creating new objects of this class and starting them.

The `IThread` class provides both GUI and non-GUI member functions. The GUI member functions are not supported on the OS/390 and OS/400 platforms. All non-GUI member functions are supported, with the following exceptions:

- `suspend()` and `resume()` are not supported

An exception of type `invalidRequest` will occur if you call either of these functions.

- Thread priority functions are not supported

The following functions through which thread priority can be manipulated have been made NO-OPs.

```
priorityLevel() const;  
virtual IApplication::PriorityClass  
priorityClass() const;  
&setPriority ( IApplication::PriorityClass priority, unsigned level );  
&adjustPriority ( int delta );
```

- Changing stack size for threads is not supported

These functions are NO-OPs. System default stack size is used for all threads.

In addition to the `IThread` class, you can use the `ICurrentThread` class to set and query attributes for the currently executing thread and wait until another thread has terminated. 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`.

All non-GUI member functions of `ICurrentThread` are supported, with the following exceptions:

```
waitForAnyThread();  
&waitForAllThreads();
```

Starting a Thread

An exception of type `invalidRequest` will occur if you call either of these functions.

Accessing the Current Thread

There is only a single object of the `ICurrentThread` class for each application, and it can be accessed using the following statement:

```
ICurrentThread& curThread = IThread::current();
```

This object accesses information held on a per-thread basis. The member also accesses some functions that can be applied only to the current thread.

Starting a Thread

Use the `IThread` class to start a thread of execution. Once started, the `IThread` object provides a means of querying and stopping the thread. The thread and the `IThread` object are independent; therefore, when the `IThread` object is destroyed, the thread is unaffected. You can start additional threads using `IThread::start`.

The function to be dispatched on a separate thread can be either a member function or a nonmember function. If you create an object of `IThread` with the function, a thread is created and dispatched immediately. Alternatively, you can create an object of the class and later dispatch it. This allows you to set arguments that affect the execution of the thread prior to dispatching.

Starting Nonmember Functions

The `IThread` class dispatches nonmember functions with either of the following two function prototypes:

```
void (_Optlink*)(void *)  
void (_System*)(unsigned long)
```

To start a thread with the default environment and default options, the following statements are needed:

```
void threadFn(void *pvParms);    //Function to run on separate thread  
void      *pv;                  //Argument for threadFn function  
  
IThread  thread(threadFn, pv);   //Dispatch thread with default environment
```

Starting a Member Function

Use the `IThread` class to start member functions. Direct support is provided for starting member functions that have no arguments, but you can also start functions that have arguments.

To start a member function that takes no arguments, use the following steps:

1. Create an object of the template class `IThreadMemberFn`.
2. Start a thread and pass the object as an argument.

The following example shows how to execute the function `AClass::longFn` on a separate thread. Create an object of the template class with the class that contains the member function. Create the object of the template class with the operator `new` function so that the object is deleted automatically when the thread ends. The two

arguments on the constructor are the object for which the member function is called and the member function itself, as shown in the following example:

```
/* function to run is ... void AClass::longFn() */
AClass object; //Object to run member function against

IThreadMemberFn<AClass> *aMemberFn =
    new IThreadMemberFn<AClass>( object
                                , AClass::longFn );
IThread thread( aMemberFn ); //Dispatch thread
```

To start a member function that takes arguments, use the following steps:

1. Derive a class from the IThreadFn class.
2. Define a constructor that takes an object of the class and the arguments you want to pass.
3. Override the ICurrentApplication::run member function to call the member function.
4. Create an object of the derived class.
5. Start a thread and pass the object as an argument.

The following example shows how to start a function:

1. Write the class declaration. The class is derived from the IThreadFn class. It has a single constructor that requires an object of the AClass class and the two parameters. The class overrides the virtual function run and calls the required member function, as shown in the following example:

```
class AClass
{
public:
    void longFn(int, IString);
    /* ... rest of class declaration ... */
};

//This class runs the member function
// AClass::longFn(...) on a separate thread
class AThreadLongFn : public IThreadFn
{
public:
    AThreadLongFn(AClass &obj, int i, IString str)
        : object( obj )
        , value( i )
        , string( str ) { }
    virtual void run() { object.longFn( value, string ); }
private:
    AClass &object;
    int value;
    IString string;
};
```

2. Create an object and dispatch it. As before, create the object using operator new so that it is deleted automatically:

```
AClass object; //Object to run member function against
int number = 6;
IString greeting( "Hello" );

/* function to run is ... void AClass::longFn(int, IString) */
//Create object
AThreadLongFn *aMemberFn = new AThreadLongFn( object, number, greeting );
IThread thread( aMemberFn ); //Dispatch thread
```

Protecting Data

If your applications have multiple threads, you typically need to serialize their access to certain resources. *Mutexes* and *semaphores* enable separate threads and processes to synchronize access to shared resources. Semaphore objects ensure that two processes do not write to the same file at the same time, and mutex objects ensure that two threads do not update static data simultaneously. The Application Support Class Library provides several classes to assist you. Use the `IPrivateResource` class to serialize access to a resource within a single process. The `ISharedResource` class provides a lock that can be used between processes.

Note: Shared locks are process scoped. In a multi-threading environment, it is the programmer's responsibility to serialize the acquisition of shared locks. That means that only one thread within a process should be waiting to acquire the shared lock at any given time.

The simplest way to serialize access to a function is to provide a static object of the `IPrivateResource` class. You can use this object in association with the `IResourceLock` class to control access. In the following example, the function guarantees that only one thread accesses it at one time:

```
static IPrivateResource resourceKey;    //Key must exist when function
                                       // called

void serializedFunction()
{
    IResourceLock resLock(resourceKey);    //Create lock
    /* ... serialized code ... */
}                                         //Lock freed with resLock destructed
```

When a thread calls `serializedFunction`, it is blocked until any other thread executing the function exits it. The `IResourceLock` class constructor takes a timeout parameter which defaults to -1 in the OS/390 environment.

Timeout value on the `lock()` member function is not supported. A default value of -1 is assumed, which means the thread must wait until the lock is obtained. This can lead to deadlock problems if the thread currently executing fails to exit.

Note: The kernel must be active to use the `lock()` member functions. In a non-OS/390 UNIX environment, all `lock()` member functions are NO-OPs.

Chapter 23. The IBM Open Class Notification Framework

This chapter provides an overview of the IBM class notification framework. You use this framework when coding with the IBM Open Class Library or to implement event and attribute notification for nonvisual parts.

Using the notification framework, registered observer objects can observe any changes being made to other objects. When one object is observing another, it receives all the notifications for every action applied to that object. The observers then select which notifications to ignore.

The notification framework contains the following entities:

- Notifier objects that support the notifier protocol defined by the `INotifier` class
- Observer objects that support the observer protocol defined by the `IObserver` class
- Notification IDs, which are defined for parts that have been enabled for event notification
- Notification event objects defined by the `INotificationEvent` class

Notifiers and Observers

Notifier objects enable other objects in the system to register dependence upon the state of the notifier objects' properties. To register dependence, objects add an observer object to the notifier object by using the following function in the `IObserver` class:

```
virtual IObserver
&handleNotificationsFor (INotifier& aNotifier,
                        const IEventData& userData = IEventData()),
```

The `IObserver` class also supports removing an observer from a notifier via the following:

```
virtual IObserver
&stopHandlingNotificationsFor (INotifier& aNotifier );
```

Notifier objects are responsible for publishing their supported notification events, managing the list of observers, and notifying observers when an event occurs. To notify observers of attribute changes or events, notifiers use the following member function defined by the `INotifier` class:

```
virtual INotifier
&notifyObservers (const INotificationEvent& anEvent) = 0;
```

The `INotifier` abstract base class defines the notifier protocol and requires its derived classes to completely implement its interface. To ensure that all notifier objects can coexist, no data is stored in any notifier object.

A notifier adds observers to an observer list and uses this list to notify observers in a first-in, first-notified manner.

The `IObserver` class defines the protocol that accepts event signals from the notifier object by overriding the member function in the `IObserver` class as follows:

```
virtual IObserver  
&dispatchNotificationEvent (const INotificationEvent&)=0;
```

Because a single list of observers is kept for each notifier, all observers in the list get called when any notification occurs within the notifier. Each observer must test to determine if a given notification event should be processed. Normally, this is done by checking notificationId in an INotificationEvent object.

Notifier objects publish the notification events that they support by providing a series of unique identifiers in their interface. These *notification IDs* are static string constants that are defined in the notifier. The string is in the form of the class name followed by the event name, such as MyString::textChanged. Each notification event provides a unique public static notification ID.

Events are typically a notification of changes in the attributes or intrinsic data that can be accessed in a notifier object. Attributes can represent any logical property of a part, such as the balance of an account, or the size of a shipment.

A *notification event* is the data provided to an observer object when a change occurs in the attributes of an object. Included in this data is the identity of the attribute being changed and the part in which the change has occurred. Also, some of the data supplied to the observer can be the actual data being changed in the notifier object.

A notification event can also include observer-specific data. The caller that registers the observer with a notifier provides this data as the userData parameter on the following call in the IObserver class:

```
virtual IObserver  
&handleNotificationsFor (INotifier& aNotifier,  
                        const IEventData& userData = IEventData()),
```

The notifier passes this data to that observer anytime it notifies the observer of an event.

Note: The notification framework in the Application Support Class Library is thread safe. However, it does not inherently support notification between threads. A notification ultimately causes a function call from notifier to observer. This does not work when there is a thread boundary between the notifier and observer. You would have to use some other means to get notifications from one thread to another. Operating system message posting is one method. You could have the second thread post messages back to the first who could then pass the notifications to the observers on its thread.

Notification Protocol

Concrete classes that inherit from the INotifier class implement its protocol. This includes the following:

- Enabling, disabling, and querying the ability to signal events. In general, notifiers are created disabled and must be enabled before they can signal events. This allows notifier objects to delay the setup to support notification until the notifier is enabled. The following member functions in the INotifier class enable you to enable and disable notification:

```
virtual INotifier  
&enableNotification (Boolean enabled = true) = 0,  
&disableNotification () = 0;
```

- Managing the collection of observers, including adding and removing observers. These are defined by the following protected members in `INotifier`:

```
virtual INotifier
&addObserver      (IObserver&      anObserver,
                  const IEventData& userData) = 0,
&removeObserver   (const IObserver& anObserver) = 0,
&removeAllObservers() = 0;
```

- Within the notifier object, calling the following member function every time an event of interest occurs:

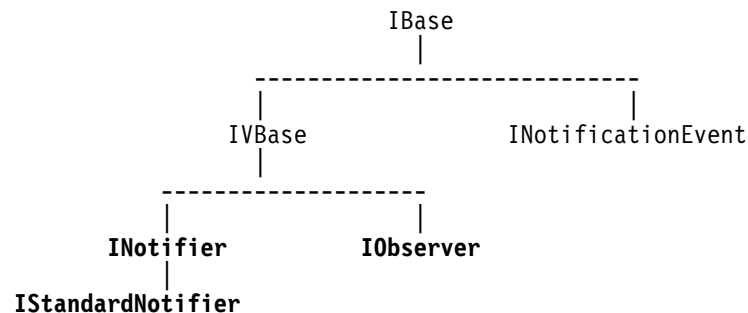
```
notifyObservers(const INotificationEvent&)
```

While the classes providing notification must call this function, in many cases it makes sense that the responsibility be delegated to another class. For instance, in the IBM Open Class Library, this responsibility is typically delegated to handler style objects.

- The protected member `INotifier::addObserver` accepts a piece of typeless data as a `const IEventData&` that is forwarded to the `IObserver` instance with any notification request. This enables a piece of data to be maintained for each instance of an observer.

The `IStandardNotifier` class provides the concrete implementation of the notifier protocol and provides the base support for nonvisual notifiers. This class inherits from a notifier class that supports registration of and notification to observer objects.

IBM C++ Notification Class Hierarchy



Within this partial hierarchy, note the following:

- The `INotifier` abstract class defines the notifier protocol.
- The `IObserver` abstract class defines the observer protocol.
- The `INotificationEvent` class implements the notification event object.
- The `IStandardNotifier` class is a concrete implementation of the notifier protocol.
- Nonvisual notifiers would normally be derived from `IStandardNotifier`.

Chapter 24. Using the Binary Coded Decimal Class

This chapter describes the `IBinaryCodedDecimal` class you use to represent numerical quantities accurately in business and commercial applications for financial calculations.

The `IBinaryCodedDecimal` class allows representation of up to 31 significant digits, including integral and fractional parts. The fractional part of a dollar can be represented accurately by two digits following the decimal point. You do not have to use floating-point arithmetic, which is more suitable for scientific and engineering computations. These computations often use numbers much larger than the largest that the `IBinaryCodedDecimal` object can store.

The same declarations and operators that you use on other data types, such as `float`, can be applied to `IBinaryCodedDecimal` objects. You can declare typedefs, arrays, and structures that have `IBinaryCodedDecimal` objects. You can apply arithmetic, relational, assignment, comma, conditional, equality, logical, primary, and unary operators on the `IBinaryCodedDecimal` object. You can pass `IBinaryCodedDecimal` objects in function calls.

Header File and Constants for IBinaryCodedDecimal

You must include this statement in any file that uses the `IBinaryCodedDecimal` class:

```
#include <idecimal.hpp>
```

The file must be included before any use of the `IBinaryCodedDecimal` object.

Constants Defined in idecimal.hpp

Table 8 lists the binary coded decimal constants that the Binary Coded Decimal Class Library defines:

Table 8. Constants Defined in idecimal.hpp

Constant Name	Description
DEC_DIG	The maximum number of significant digits that <code>IBinaryCodedDecimal</code> can hold.
DEC_MIN	The minimum value that <code>IBinaryCodedDecimal</code> can hold.
DEC_MAX	The maximum value that <code>IBinaryCodedDecimal</code> can hold.
DEC_EPSILON	The smallest incremental or decremental value that <code>IBinaryCodedDecimal</code> can hold.
DFT_DIG	The default number of digits (15) for the default constructor.
DFT_PREC	The default number of precision (5) for the default constructor.
DFT_LNG_DIG	The default number of digits (20) for a long type.

Constructing IBinaryCodedDecimal Objects

You can use the `IBinaryCodedDecimal` constructor to construct `IBinaryCodedDecimal` objects or arrays of `IBinaryCodedDecimal` objects. The following example shows how to construct an `IBinaryCodedDecimal` object to have a value (12) with `DFT_LNG_DIG`, number of digits (20) and number of precisions (0):

```
IBinaryCodedDecimal a(12L);
```

The following example shows how to construct an `IBinaryCodedDecimal` object to have a value `INT_MAX` with number of digits (16) and number of precisions (5):

```
IBinaryCodedDecimal b(16,5,INT_MAX);
```

IBinaryCodedDecimal Input and Output

You can use the input and output operators for the I/O Stream Library to perform input and output operations on `IBinaryCodedDecimal`. See Part 2, "The I/O Stream Class Library" on page 23 for more in-depth information on using the I/O Stream Library.

Mathematical Operators for IBinaryCodedDecimal

The `IBinaryCodedDecimal` class defines a set of operators with the same precedence as the corresponding real operators. With these operators, you can code expressions on `IBinaryCodedDecimal` objects such as the expressions shown in the example below:

```
IBinaryCodedDecimal value1("123.78");
IBinaryCodedDecimal value2("345.12");
IBinaryCodedDecimal value3("77.457");
IBinaryCodedDecimal Sum, Average;

Sum = value1 + value2;
Sum += Sum + value3; // Sum should have value 546.357
Average = Sum / 3; // Average should have value 182.119
```

If accuracy of `IBinaryCodedDecimal` is important, use the `char *` contractor instead of the floating type contractor. For example, use

```
IBinaryCodedDecimal accurateBCD("12345.6789");
// this will store exactly 12345.6789
```

instead of

```
IBinaryCodedDecimal roughBCD(12345.6789);
// this will store something close to 12345.6789
// (might be 12345.678899999999..., depends on the
// floating type representation)
```

Relational Operators

You can use the relational operators `<` `>` `<=` `>=` for `IBinaryCodedDecimal` objects and compare `IBinaryCodedDecimal` objects with other arithmetic types (integer, float, double, and long double):

```
IBinaryCodedDecimal BCD_1(15);
IBinaryCodedDecimal BCD_2(-15);

if (BCD_1 < BCD_2)
...
```

Equality Operators

You can use equality operators with `IBinaryCodedDecimal` objects to compare `IBinaryCodedDecimal` objects for equality:

```
IBinaryCodedDecimal BCD_1(15);
IBinaryCodedDecimal BCD_2(-15);

if ( BCD_1 != BCD_2 )
...
```

Converting IBinaryCodedDecimal Objects

The `IBinaryCodedDecimal` class defines a set of conversion operators. With these operators you can convert `IBinaryCodedDecimal` objects to other data types.

IBinaryCodedDecimal Object to a IBinaryCodedDecimal Object

If the value of an `IBinaryCodedDecimal` object that is to be converted to another `IBinaryCodedDecimal` object is not within the range of values that can be represented exactly, the value of the `IBinaryCodedDecimal` object to be converted is truncated. If truncation occurs in the fractional part, there is no exception raised. If assignment causes truncation in the integral part, then there is an exception in which a `IDecimalDataError` object is thrown. This exception occurs when an integral value is lost during conversion to a different type, regardless of what operation requires the conversion:

```
IBinaryCodedDecimal targ_1(4,2);
IBinaryCodedDecimal targ_2(4,2);
IBinaryCodedDecimal op_1("1234.56");
IBinaryCodedDecimal op_2("12.34");

targ_1=op_1; // An exception is generated because the integral
             // part is truncated; targ_1="34.56".

targ_2=op_2; // No exception is generated because neither the
             // integral nor the fractional part is truncated;
             // targ_2="12.34".
```

An exception occurs on assignment to a smaller target only when the integral part is truncated.

When one `IBinaryCodedDecimal` object is assigned to another `IBinaryCodedDecimal` object with a smaller precision, the result is truncation of the fractional part:

```
IBinaryCodedDecimal x("123.4567");
IBinaryCodedDecimal y(7,1);

y = x;    // y = ("123.4")
```

When one `IBinaryCodedDecimal` object is assigned to another `IBinaryCodedDecimal` object with a smaller integral part, the result is truncation of the integral part. An exception occurs:

```
IBinaryCodedDecimal x("123456.78");
IBinaryCodedDecimal y(5,2);

y = x;    // y = ("456.78")
```

When one `IBinaryCodedDecimal` object is assigned to another `IBinaryCodedDecimal` object with a smaller integral part, and smaller precision, the result is truncation of the integral, and fractional parts. An exception occurs:

IBinaryCodedDecimal Object Exceptions

```
IBinaryCodedDecimal x("123456.78");
IBinaryCodedDecimal y(4,1);

y = x; // y = ("456.7")
```

Number of Digits of an IBinaryCodedDecimal Object

When you use the member function `digitsOf()` with an `IBinaryCodedDecimal` object, you can find out the total number of digits `n` in an `IBinaryCodedDecimal` object:

```
int n;
IBinaryCodedDecimal x(5, 2);
n = x.digitsOf(); // the result is n=5
```

Precision of an IBinaryCodedDecimal Object

When you use the member function `precisionOf()` with an `IBinaryCodedDecimal` object, you can find out the number of decimal digits `p` in an `IBinaryCodedDecimal` object:

```
int p;
IBinaryCodedDecimal x(5, 2);
p=x.precisionOf(); // The result is p=2
```

IBinaryCodedDecimal Object Exceptions

The `IDecimalDataError` exception class is thrown whenever the integral part is truncated as the result of any mathematical operation.

Chapter 25. Using the Decimal Class

This chapter describes the decimal class you use to represent numerical quantities accurately in business and commercial applications for financial calculations.

OS/390 C++ supports the decimal data type through the `IBinaryCodedDecimal` class as well as the decimal class. Use the decimal class to improve the performance of your applications relative to using the `IBinaryCodedDecimal` class. The decimal class is compatible with the decimal data type in C. This class permits you to represent up to 31 significant digits, including integral and fractional parts.

You can declare typedefs, arrays, and structures that have decimal objects. You can apply arithmetic, relational, assignment, equality, and unary operators on the decimal object. You can pass decimal objects in function calls.

Header File

You must include this statement in any file that uses the decimal class:

```
#include <idecimal.hpp>
```

The file must be included before any use of the decimal object.

Constructing Decimal Objects

You can use the decimal constructor to construct decimal objects or arrays of decimal objects. Use the template specifier `decimal<w,p>` to declare decimal objects. The template specifier `decimal<w,p>` designates a decimal number with *w* digits, and *p* decimal places. In the specifier, *w* is the total number of digits for the integral and decimal parts combined. *p* is the number of digits for the decimal part only. For example, `decimal <5,2>` represents a number, such as 123.45, where *w*=5 and *p*=2. Specifying the value for *p* is optional. If omitted, OS/390 C++ creates a default value of 0 for *p*.

In the specifier, *w* and *p* have a range of allowed values according to the following rules:

$$1 \leq w \leq 31$$

$$0 \leq p \leq w$$

You can construct a decimal object using an integer, a `char *`, a decimal object, or another `IBinaryCodedDecimal` object. The following example shows how you can construct a decimal type:

```
decimal<10,2>  x("4.67");           // char *
decimal<5,0>   y(7);                 // integer
decimal<5>     z=y;                   // another decimal object
decimal<18,10> *ptr;                 // pointer
decimal<8,2>   arr[100];             // array
IBinaryCodedDecimal a(12)            //another IBinaryCodedDecimal object
decimal<10,3>  b(a);
```

In the previous example:

- *x* has a value of +4.67.

- y and z have a value of +7.
- ptr is a pointer to type decimal<18,10> .
- arr is an array of 100 elements, where each element is of type decimal<8,2>.
- b has the value of the decimal object a, +12.

Decimal Class Input and Output

You can use the input and output operators for the I/O Stream Library to perform input and output operations on decimal. See Part 2, “The I/O Stream Class Library” on page 23 for more in-depth information.

Operators for Decimal Class

Arithmetic Operators

The decimal class defines a set of arithmetic operators with the same precedence as the corresponding non-overloaded operators. With these operators, you can perform arithmetic calculations between two decimal objects, or between a decimal object and an integer.

```
decimal<5,2>      x("9.45");
decimal<8,3>      y(-3);
decimal <20,13>   sum = x + y;
```

Intermediate Results

Use one of the following tables to calculate the size of the result. The tables summarize the intermediate expression results with the four basic arithmetic operators when applied to the decimal types. Most of the time, you can use Table 9 to calculate the size of the result. It assumes no overflow. If overflow occurs, use Table 10 on page 245 to determine the resulting type.

Both tables assume the following:

- x has type decimal<w₁,p₁>
- y has type decimal<w₂,p₂>
- decimal<w,p> is the resulting type

Table 9. Intermediate Results (without overflow in w or p)

Expression	<w,p>
x * y	w = w ₁ + w ₂ p = p ₁ + p ₂
x / y	w = 31 p = 31 - ((w ₁ - p ₁) + p ₂)
x + y	p = max(p ₁ , p ₂) n; = max(w ₁ - p ₁ , w ₂ - p ₂ ;) + p + 1
x - y	same rule as addition

You can use Table 10 on page 245 to calculate the size of the result, whether there is an overflow or not.

Table 10. Intermediate Results (in the general form)

Expression	<w,p>
$x * y$	$w = \min(w_1 + w_2, 31)$ $p = \min(p_1 + p_2, 31 - \min((w_1 - p_1) + (w_2 - p_2), 31))$
x / y	$w = 31$ $p = \max(31 - ((w_1 - p_1) + p_2), 0)$
$x + y$	ir $= \min(\max(w_1 - p_1, w_2 - p_2) + 1, 31)$ $p = \min(\max(p_1, p_2), 31 - ir)$ $w = ir + p$
$x - y$	same rule as addition

Relational Operators

You can use the relational operators `<` `>` `<=` `>=` for decimal objects. You can compare two decimal objects, or a decimal object with an integer.

```
decimal<5,2>    x("10.0");
decimal<8,3>    y("-2.3");
if (x < y)
...
```

Equality Operators

You can use the equality operators `!=` `==` for decimal objects. You can compare two decimal objects or a decimal object with an integer for equality.

```
decimal<5,2>    x(15);
decimal<5,2>    y(-15);
if ( x != y )
...
```

Converting Decimal Objects

The decimal class defines a set of conversion operators and functions. With these operators and functions, you can convert decimal objects to and from other data types.

If the value that is to be converted is not within the range of values that can be represented exactly, OS/390 C++ truncates this value. If truncation occurs in the fractional part, OS/390 C++ does not raise an exception. If assignment causes truncation in the integral part, OS/390 C++ raises an exception. This exception occurs when an integral value is lost during conversion to a different type, regardless of the operation that requires the conversion.

Decimal Object to a Decimal Object

The following is an example of converting a decimal object to another decimal object:

```
decimal <5,2>    x(3);
decimal <31,15> y;
y = x;
```

Decimal Object to an IString Object

OS/390 C++ provides a member function, `asString()`, to convert a decimal object to an `IString` object. The following is an example of such a conversion:

```
decimal<5,2> x("3.46");  
IString y = x.asString();
```

Decimal Object from a char * Type

The following is an example of converting a `char *` type to a decimal object:

```
char * x = "1234.5";  
decimal<5,2> y;  
y = x;
```

Decimal Object from an Integer Type

The following is an example of converting an integer to a decimal object:

```
int x=3;  
decimal<3,1> y=x;
```

Decimal Object to and from IBinaryCodedDecimal Object

The following is an example of converting a decimal object from an `IBinaryCodedDecimal` object:

```
IBinaryCodedDecimal y(12);  
decimal<5,2> x(y);
```

OS/390 C++ provides a member function, `asBCD()`, to convert a decimal object to an `IBinaryCodedDecimal` object. The following is an example of such a conversion:

```
decimal<5,2> x("3.46");  
IBinaryCodedDecimal y = x.asBCD();
```

Number of Digits in a Decimal Object

When you use the member function `digitsOf()` with a decimal object, you can find out the total number of digits `w` in a decimal object:

```
int w;  
decimal<5,2> x;  
w = x.digitsOf(); // the result is w=5
```

Precision of a Decimal Object

When you use the member function `precisionOf()` with a decimal object, you can find out the number of decimal digits `p` in a decimal object:

```
int p;  
decimal<5,2> x;  
p=x.precisionOf(); // The result is p=2
```


Decimal Object Exceptions

OS/390 C++ decimal instructions produce the following exceptions:

- Data exception (interrupt code hex '7')

This may be caused by invalid sign or digit codes in a packed decimal number operated on by packed decimal instructions.

- Decimal-overflow exception (interrupt code hex 'A')

This exception may be caused when nonzero digits are lost because the destination field in a decimal operation is too short to contain the result.

CEE3210S The system detected a Decimal-overflow exception.

- Decimal-divide exception (interrupt code hex 'B')

This exception may be caused when, in decimal division, the divisor is zero, or the quotient exceeds the specified data-field size. The decimal divide is indicated if the sign codes of both the divisor and dividend are valid, and if the digit or digits used in establishing the exception are valid.

Note: The following unhandled divide message does not distinguish between a decimal-divide condition and a fixed divide-by-zero condition:

CEE3211S The system detected a Decimal-divide exception.

Both are mapped into the same error message.

- SIGFPG exception

During the conversion of `char *` to the decimal object, there is a possibility that the value of the integer part cannot be represented by the decimal type. In that case, the result of the conversion is undefined and OS/390 C++ raises a SIGFPG exception.

Part 5. Glossary, Bibliography and Index

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.

Decimal Object Exceptions

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, intrapartition, and indirect destination. Extrapartition entries address data sets external to the CICS region. Intrapartition destination entries contain the information required to locate the queue in the intrapartition data set. Indirect destination entries contain the information required to locate the queue in the intrapartition 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 ≤ 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 xv 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 `fdopen()` 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

A

- absolute value of complex numbers 11
- abstract Collection Classes
 - based-on concept 122
 - class hierarchy 85
 - cursor class used with 94
 - key collection
 - restriction on replacing elements 93
 - naming convention 86
 - polymorphism 131
 - relationship to other classes 84
- accessing elements 80, 94
- add() Collection Class function
 - behavior of 91
 - example of behavior 91
 - properties of 90
 - role of 90
- addAsFirst() Collection Class function 91
- addAsLast() Collection Class function 91
- addAsNext() Collection Class function 91
- addAsPrevious() Collection Class function 91
- adding elements to collections
 - effect on cursors 94
 - overview 90—91
- addition of complex numbers 11
- addOrReplaceElementWithKey() Collection Class function 90
- allElementsDo() Collection Class function 97, 98
- anonymous streams 29
- Application Support classes
 - introduction 191
 - IBase class 191
 - IBaseErrorInfo class 191
 - IBuffer class 191
 - IDBCSBuffer class 191
 - IVBase class 194
- applicator 67
- assignment (Collection Class Library)
 - using member functions 102
 - using operation classes 106
 - using separate functions 103
- AT&T C++ Language System Release 1.2
 - history of class libraries 1
- auxiliary class 77, 83

B

- bag
 - description 74
 - implementation variant explained 122
 - properties of 78

- IBase class 194
- base() streambuf function 31
- based-on concept in Collection Class Library
 - overview 121—122
- binary conversion in IString class 208
- bounded collections 100

C

- case change of IString objects 211
- cerr predefined stream 28
- IChildAlreadyExistsException 145
- children of a tree node 82
- cin predefined stream 28, 35
- class
 - categories of Collection Classes 83—87
 - general types of Collection Classes 77
 - hierarchy
 - abstract collections 85
 - Application Support classes 191
 - Collection Class Library 86
 - Complex Mathematics 1
 - I/O Stream Classes 27
 - illustrations 1
- clog predefined stream 28
- collection
 - copying 99
 - cursor association 93
 - iterating over 96—99
 - modifying 90—93
 - referencing 99
 - using polymorphism with 131
- Collection Class Library
 - categories of classes 83—87
 - implementation strategy 83
 - reasons for using 73
 - steps for using 89
 - structure of classes 84
 - types of collections 77
- ICollectionLockException 146
- ICollectionResourceException 146
- ICollectionUnlockException 147
- compare() function
 - Collection Class Library
 - using operation classes 106
 - using separate functions 103
- comparison
 - of IDate objects 226
 - of ITime objects 228
 - of ITimeStamp objects 230
- complex class
 - conversion functions 18

- complex class (*continued*)
 - mathematical functions 16
 - mathematical operators 14
 - review of complex numbers 11
 - trigonometric functions 17
- Complex Mathematics Library 11, 12
- complex.h header file 12
- concatenation of IString objects 204
- concrete classes
 - cursors with 94
 - relationship to other classes 84
- concrete implementations 85, 122
- conjugates of complex numbers 11
- constant iterator class 97—99
- constants defined in ideoimal.hpp 239
- constructors
 - Collection Class Library
 - errors 182
 - restriction on defining 102
 - IDate class 225
 - IString class 201
 - IStringTest class 213
 - ITime class 227
 - ITimeStamp class 229
- containment function 81
- conversion 245, 246
 - decimal object and IBinaryCodedDecimal object 246
 - decimal object from a char * type 246
 - decimal object from an integer type 246
 - decimal object to a decimal object 245
 - decimal object to an IString object 246
 - decimal objects 245
 - IBinaryCodedDecimal object to a IBinaryCodedDecimal object 241
 - IBinaryCodedDecimal objects 241
- conversion functions
 - complex class 18
 - IString class 201, 208
- copying
 - collections 99
 - IString class 202
- cout predefined stream 28, 38
- ICursorInvalidException 145
- cursors
 - accessing elements with 94
 - association with a collection 93
 - classes 94
 - classes, abstract 94
 - description 93—95
 - effect of replacing elements 93
 - iteration 96—97
 - locating elements with 94
 - properties that may cause an exception 145
 - reasons for using 94
 - removing elements with 92

- cursors (*continued*)
 - unexpected results 177
 - validity 94, 145
- customizing an implementation 121—130

D

- Data Type Classes
 - IDate class 199—212
 - IDate class 225—230
 - IString class 199—212
 - IStringTest class 212—213
 - ITime class 225—230
 - ITimeStamp class 225—230
- IDate class 225—230
- decimal 243, 247
 - constructing objects 243
 - exceptions 247
- decimal class 243, 244
 - arithmetic operators 244
 - input and output 244
- decimal class representation 243
- decimal conversion in IString class 208
- decimal object 246
 - asBCD 246
 - asString 246
 - digitsof 246
 - precisionof 246
- default classes in Collection Class Library
 - instantiation 89
 - introduction 85
 - naming convention 86
 - relationship to other classes 84
 - strategy for using 83
 - tutorial on using 173
- default constructors 177
- deleting substrings of IString objects 206
- deque 77, 82
- destructors
 - Collection Class Library
 - restriction on defining 102
- diluted implementation 124, 125
- diluted table 124
 - reasons for using 124
- dispatchNotificationEvent, overview 235

E

- eback() streambuf function 31
- ebuf() streambuf function 31
- egptr() streambuf function 31
- element equality 78, 79—81
- elementAt() function
 - accessing elements with 94
 - replacing elements using 95
 - role in Collection Class Library 93

- elementAt() function (*continued*)
 - versions of 95
- elements in Collection Class Library
 - accessing 80, 94
 - adding 90—91
 - effect on cursors 94
 - functions
 - errors 178
 - introduction 101
 - methods for providing 102
 - relationship to derived classes 109
 - using element operation classes 105
 - using member functions 102
 - using pointers with 111
 - using separate functions 103
 - iterating 96—99
 - locating 90, 94
 - See also* locate... functions
 - occurrence 92
 - operation classes 105
 - polymorphism 131
 - removing 91—92
 - effect on cursors 94
 - replacing 93
 - using elementAt() function 95
 - value 92
- elementWithKey() Collection Class function 95
- EmptyException 145
- endl manipulator 40
- epptr() streambuf function 31
- equality operators for decimal objects 245
- equality operators for IBinaryCodedDecimal 241
- equality relation 79
- equality sequence 76, 78
- equality test
 - in Collection Class Library
 - using member functions 102
 - using operation classes 106
 - using separate functions 103
 - in complex class 15
- error
 - determination in Collection Class Library 177
 - handling
 - by math.h for complex class 20
 - stream input 54
- examples
 - machine-readable xxiii
 - naming of xxiii
 - softcopy xxiii
- exception
 - IAccessError exception 215
 - IAssertionFailure exception 215
 - IChildAlreadyExistsException 145
 - ICollectionLockException 146
 - ICollectionResourceException 146
 - ICollectionUnlockException 147

- exception (*continued*)
 - ICursorInvalidException 145
 - IDeviceError exception 215
 - IEmptyException 145
 - IException class 147
 - IFullException 100, 146
 - hierarchy 147
 - IDecimalDataError exception 215
 - IdenticalCollectionException 146
 - in Collection Class Library 143—148
 - InvalidParameter exception 215
 - InvalidReplacementException 146
 - InvalidRequest exception 215
 - IKeyAlreadyExistsException 146
 - INotBoundedException 100, 146
 - INotContainsKeyException 146
 - IOutOfMemory exception 215
 - IOutOfSystemResource exception 215
 - IOutOfWindowResource exception 215
 - IPositionInvalidException 146
 - IPreconditionViolation 147
 - IResourceExhausted exception 147, 215
 - IRootAlreadyExistsException 146
 - tracing 177, 181
 - violated precondition in Collection Class Library 143
- IException class 147, 177, 215—221
 - trace function 181
- Exception Classes
 - IException class 215—221
 - ITrace class 222—224
- extraction operator
 - See* operator >>

F

- file input 42
- file output 45
- filebuf class
 - header files 28
 - moving through a file 48
- filters in I/O Stream Classes 47
- firstElement() Collection Class function 95
- flat collection classes
 - overview 77—81
 - with restricted access 81
- flush manipulator 40
- forlCursor Collection Class Macro 96
- format state
 - mutually exclusive flags 58
- formatting
 - of IString objects 211
 - of output streams 56
- fstream class
 - assigning to cin and cout 47
 - file input 42
 - file output 45

fstream class (*continued*)
 header files 28
IFullException 100, 146

G

get pointer 31
getline() istream function 36
gptr() streambuf function 31
Gregorian calendar 225
Guard objects
 description 139
 usage 139

H

handleNotificationsFor, overview 235
hashing
 description 129
 restriction on replacing elements 93
 restrictions on defining 103
 using operation classes 106
 using separate functions 103
header files
 I/O Stream Classes 28
heap
 description 76
 properties of 78
 replacing elements 93
hexadecimal conversion in IString class 208
hierarchy
 See class — hierarchy

I

Note: Most classes beginning with an uppercase 'I' are indexed under their second letter.

I/O Stream Classes
 class hierarchy 26
 header files 28
 predefined streams 28
 stream buffers 30
IBinaryCodedDecimal 239
 constants 239
 constructing objects 240
 exceptions 242
 input and output 240
 operators 240
IBinaryCodedDecimal class representation 239
IBinaryCodedDecimal object
 digitsof 242
 precisionof 242
idecimal.hpp header file 239, 243
IIdenticalCollectionException 146
ifstream class
 file input 42

ifstream class (*continued*)
 header files 28
imaginary part of a complex number 11
implementation in Collection Class Library
 basic 121
 concrete 85
 instantiating the default 89
 provided by Collection Class Library 122
 replacing the default 121
 tailoring 121—130
implementation variant
 choosing 122
 features of 123
 provided by Collection Class Library 122
indexing of strings 200
inequality operator for complex class 15
inheritance in Collection Class Library 131
INotificationEvent class overview 236
Notifier class overview 235
input
 correcting errors 54
 from files 42
 from standard input 35
 IString class 203
 white space in 36
inserting substrings into IString objects 206
insertion operator
 See operator <<
InvalidReplacementException 146
ioanip.h header file 28
ios class
 header files 28
iostream class
 header files 28
iostream Library 1
 See also 'I/O Stream', at start of 'I' entries
istream_withassign class
 example of using 47
 header files 28
istream.h header file
 classes defined 28
IPrivateResource 234
IResourceLock 234
is... methods of IString class 209
isFull() Collection Class function 100
ISharedResource 234
istream class
 header files 28
 input operator
 for class types 50
 multiple types in an input statement 35
 pointers to char 36
 white space 36
istream_withassign class
 header files 28

- IString... classes
 - See entries for IString... under S
- istream class
 - header files 28
- isValid() function
 - limitation 94
 - role of 145
- iteration
 - over collections 96—99
 - restrictions 96
 - using exceptions 144
- iterator class 97—99
- IThread 231
- IThreadFn 232
- IThreadMemberFn 232

J

- Julian date format 225

K

- key access
 - basic properties of flat collections 78
 - description 79
 - errors 177, 179
 - overview 79—81
 - restriction on defining 102
 - using operation classes 106
 - using separate functions 103
- key bag 74, 78, 91
- key collection 93
 - restriction on replacing elements 93
- key equality 79—81
- key set
 - adding elements 91
 - description 75
 - properties of 78
- key sorted bag 74, 78
- key sorted set 75, 78
- key-type functions
 - errors 177, 178, 180
 - global 180
 - introduction 101
 - methods for providing 102
 - relationship to derived classes 109
 - using element operation classes 105
 - using member functions 102
 - using pointers with 111
 - using separate functions 103
- IKeyAlreadyExistsException 146

L

- linked implementation 94

- list 124
 - description 124
- locateOrAddElementWithKey() Collection Class
 - function 90
- locating elements 94
- lowercase and IString objects 211

M

- macros for the exception classes 218
- magnitude of complex numbers 11
- managed pointer class 78, 83
 - relationship to derived classes 109
- map 75
- mathematical functions for complex class 16
- matherr() library function 20
- maxNumberOfElements() Collection Class
 - function 100
- MBCS 195
- memory management
 - restriction on defining 103
 - using operation classes 106
 - using separate functions 103
- modifying a collection 90—93
- multiple collections 78, 81
- multiple inheritance in Collection Class Library 131
- multiple-byte character set 195, 200
- multiplication of complex numbers 11
- multithreading 231
- mutual-exclusive semaphore 234
- mutually exclusive format flags 58

N

- n-ary tree class 82—83
- naming conventions 86
- National Language Support (NLS) 195
- newCursor() function
 - abstract classes 94
- NLS 195
- node of a tree 82
- nonmember functions, starting 232
- INotBoundedException 100, 146
- INotContainsKeyException 146
- notification class hierarchy 237
- notification ID overview 236
- notification protocol description 236
- notifications
 - description 135
 - support for 135
- notifier protocol overview 235
- notifyObservers, overview 235
- null character 204
- numeric conversion in IString class 208

O

- object definition, default implementation 89
- observer protocol overview 235
- obsolete functions 1
- ofstream class
 - file output 45
 - header files 28
- operations classes
 - using 105—109
- operator +
 - complex class 11
 - IString class 204
- operator <
 - Collection Class Library 102
- operator <<
 - defining for class types 52
 - in I/O Stream Classes 25
 - ostream class 66
 - IString class 203
 - ITime class 228
- operator =
 - Collection Class Library 102
- operator ==
 - Collection Class Library 102
 - complex class 15
- operator >>
 - defining for class types 50
 - in I/O Stream Classes 25
 - istream class 66
 - multiple types in an input statement 35
 - pointers to char 36
 - IString class 203
- ordered collection
 - multiple inheritance 131
 - removing an element 92
- ordering relation
 - as a collection property 78
 - possible orderings of collections 79
 - restriction on replacing elements 93
 - sorted collections 79
 - using member functions 102
- ostream class
 - header files 28
 - output operator
 - for class types 52
 - multiple types in an output statement 39
- ostream_withassign class
 - header files 28
- ostrstream class
 - header files 28
- IOutOfMemory exception 146
- output
 - to files 45

P

- padding IString objects 211
- parameterized manipulators
 - and simple manipulators 65
 - example 68
 - for your own types 67
 - introduction 65
- parent in a tree 82
- pbase() streambuf function 31
- pointer class
 - managed
 - See managed pointer class
 - relationship to derived classes 109
 - role of 78
 - using with element and key-type functions 111
- polar representation of complex numbers 11
- polymorphism 83, 131
 - using pointers to elements 111
- positioning property 93
- IPositionInvalidException 146
- pptr() streambuf function 31
- precondition
 - violated 143—145
- IPreconditionViolation exception 147
- predefined streams
 - assigning fstream objects to 47
 - defined by iostream.h 28, 35, 38
- predicate functions in Collection Class Library 92
- priority queue 77, 82
- problem determination
 - Collection Class Library 177
- protecting data 231, 234
- put pointer 31
- putback 31

Q

- queue 77
- queue collection 82

R

- real part of a complex number 11
- reference class
 - naming convention 86
 - relationship to other classes 84
- referencing a collection 99
- relation 75
 - sorted relation 76
- relational operators for decimal class 245
- relational operators for IBinaryCodedDecimal 240
- remove() function
 - Collection Class Library
 - behavior of 92
 - role of 90

- removeAll() Collection Class function 92
- removeFirst() Collection Class function 92
- removeLast() Collection Class function 92
- removing elements
 - See also* remove... functions for collections
 - effect on cursors 94
 - overview 91—92
- replace() Collection Class function 90, 93
- replaceAt() function
 - role of 93
- replacing
 - substrings of IString objects 206
- replacing elements 93
 - See also* replace... functions for collections
 - using elementAt() function 95
- IRootAlreadyExistsException 146

S

- semaphore 234
- separate functions in Collection Class
 - Library 103—104
- sequence 76
- sequence as list
 - template with element operation class 105
- sequential collections
 - replacing elements 93
- serializedFunction 234
- set 74
- simple manipulators 65
- sorted bag 74, 78
- sorted collections
 - multiple inheritance 131
 - ordering relation 79
- sorted map
 - description 75
 - properties of 78
 - restrictions for adding elements 90
- sorted relation
 - properties of 78
- sorted set 75, 78
- stack 76, 82
- standard error 38
- standard input 35
- standard output 38
- stderr 38
- stdin 35
- stdiobuf class
 - header files 28
- stdiostream class
 - header files 28
- stdiostream.h 28
- stdout 38
- stopHandlingNotificationsFor, overview 235
- stream buffers
 - definition 30

- stream buffers (*continued*)
 - implementation 30
 - purpose 30
- Stream Library 1
- stream.h header file 28
- streambuf class
 - header files 28
 - member functions 31
- string buffers 200
- IString class
 - binary conversion 208
 - concatenating objects of 204
 - decimal conversion 208
 - deleting a substring 206
 - formatting 211
 - hexadecimal conversion 208
 - indexing of strings 200
 - inserting a substring 206
 - is... methods of IString class 209
 - numeric conversion 208
 - padding 211
 - replacing a substring 206
 - string length 207
 - testing characteristics of IString objects 209
 - using 199—212
 - word count 207
- string input 36
- IStringTest class 212—213
- IStringTestMemberFn class 213
- stripping blanks from IString objects 211
- strstream class
 - header files 28
 - possible uses of 63
- strstream.h 28
- strstreambuf class
 - header files 28
- substrings
 - creating from IString objects 202
 - deleting in IString objects 206
 - finding within IString objects 204
- subtraction of complex numbers 11
- system failures 144
- system restrictions 144

T

- table 124, 125
- table implementation 124, 125
- table sequence 124
- tabular implementation 94
- tailoring an implementation 121—130
- templates
 - arguments
 - declaration errors 177, 181, 182
 - linking with 183
 - operation class inheritance 106

- thread 231
- thread safety
 - collection classes
 - description 139
 - Guard objects 139
 - description 7
 - levels of 7
 - reasons for 7
- ITime class 225—230
- ITimeStamp class 225—230
- ITrace class 215—224
- trace macros 222
- tree 77, 82—83
- trigonometric functions for complex class 17
- tutorials 149—175
- typed implementation class
 - naming convention 86
 - purpose 85
- typeless implementation class
 - purpose 86
 - relationship to other classes 84

U

- ultimate consumer 30
- ultimate producer 30
- unbounded collections 100
- unique collections
 - adding elements 91
 - compared to multiple collections 78
 - description 81
- UNIX System Laboratories C++ Language System 1
- unordered collections
 - characteristics 79
 - cursor iteration drawbacks 97
- uppercase and IString objects 211
- user-defined input operator 35, 50
- user-defined output operator 39, 52

V

- variant classes
 - See also* implementation variant
 - description 85
 - naming convention 86
 - strategy for using 83
 - tailoring a collection with 121—130
- IVBase class 194
- void* type 92

W

- white space
 - in IString objects 211
 - in string input 36

Communicating Your Comments to IBM

OS/390
C/C++
IBM Open Class Library User's Guide

Publication No. SC09-2363-03

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 User's Guide

Publication No. SC09-2363-03

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



SC09-2363-03

