Note!

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

First Edition (November 1996)

This edition applies to Version 3 Release 7 Modification Level 0, of VisualAge for C++ for AS/400 (Program 5716-CX5) and to all subsequent releases and modifications until otherwise indicated in new editions. Make sure you are using the correct edition for the level of the product.

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

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

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

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

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

© **Copyright International Business Machines Corporation 1995, 1996. All rights reserved.**
Note to U.S. Government Users — Documentation related to restricted rights — Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corp.

Contents

Notices xi
Programming Interface Information xi
Trademarks and Service Marks xii

About This Book xiii
Who Should Use This Book xiii
How to Use This Book xiv
How to Find Class or Function Descriptions xiv
A Note about Examples xv
How to Get Help xv
Getting Help Inside VisualAge for C++ for AS/400 xvi
Getting Help from the Command Line xvi
Getting Help for a Keyword or Construct xvi
Online Documents Available in VisualAge for C++ for AS/400 xvii

Chapter 1. Introduction to IBM Open Class Library 1
History of IBM Open Class Library 1
Hierarchies of the Class Libraries 2
Including IBM Open Class Library 5

Part 1. Complex Mathematics 7

Chapter 2. Using the Complex Mathematics Classes 9
Review of Complex Numbers 9
Header Files and Constants for complex and c_exception 10
Constructing complex Objects 11
Complex Mathematics Input and Output 11
Mathematical Operators for complex 12
Equality and Inequality Operators Test for Absolute Equality 13
Assignment Operators Do Not Produce an lvalue 14
Friend Functions for complex 15
Mathematical Functions for complex 15
Trigonometric Functions for complex 16
Magnitude Functions for complex 16
Conversion functions for complex 17
Using the c_exception Class to Handle Complex Mathematics Errors 17
Defining a Customized complex_error Function 18
Errors Handled Outside of the Complex Mathematics Library 19
An Example of Using the Complex Mathematics Library 19

Part 2. The I/O Stream Library 23

Chapter 3. Introduction to the I/O Stream Classes 25
The I/O Stream Classes and stdio.h 25

© Copyright IBM Corp. 1995, 1996

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 . . . 29
Anonymous Streams . . . 30
Stream Buffers . . . 31
What Does a Stream Buffer Do? . . . 31
Why Use a Stream Buffer? . . . 31
How Is a Stream Buffer Implemented? . . . 32
Format State Flags . . . 33

Chapter 4. Getting Started with the I/O Stream Library . . . 35
Receiving Input from Standard Input . . . 35
Multiple Variables in an Input Statement . . . 36
String Input . . . 36
White Space in String Input . . . 37
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 . . . 40
Flushing Output Streams with endl and flush . . . 40
Placing endl or flush in an Output Stream . . . 41
Parsing Multiple Inputs . . . 42
Opening a File for Input and Reading from the File . . . 43
Constructing an fstream or ifstream Object for Input . . . 43
Reading Input from a File . . . 45
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 . . . 49
Using the cin Stream in a Class Input Operator . . . 50
Displaying Prompts in Input Operator Code . . . 51
Defining an Output Operator for a Class Type . . . 51
Class Output Operators and the Format State . . . 52
Correcting Input Stream Errors . . . 53
Changing the Formatting of Stream Output . . . 55
ios Methods and Manipulators . . . 56
Using setf, unsetf, and flags . . . 56
Changing the Notation of Floating-Point Values . . . 58
Changing the Base of Integral Values . . . 59
Setting the Width and Justification of Output Fields . . . 60
Defining Your Own Format State Flags . . . 61
Using the strstream 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 . . . 80
  Equality for Keys and Elements . . . 80
  Uniqueness of Entries . . . 82
Restricted Access . . . 82
Trees . . . 83
Auxiliary Classes . . . 84
The Overall Implementation Structure . . . 84
  Categories of Classes . . . 85
  Default Classes . . . 87
  Variant Classes . . . 87
  Collection Class Hierarchy . . . 87
  Typed and Typeless Implementation Classes . . . 87
  Class Template Naming Conventions . . . 88
Binding to the Collection Classes . . . 89

Chapter 8. Instantiating and Using the Collection Classes . . . 91
Instantiation and Object Definition . . . 91
Adding, Removing, and Replacing Elements . . . 92
  Adding Elements . . . 92
  Removing Elements . . . 93
  Replacing Elements . . . 95
Cursors . . . 95
  Using Cursors for Locating and Accessing Elements . . . 96
Iterating over Collections . . . 98
  Iteration Using Cursors . . . 98
  Iteration Using allElementsDo . . . 99
  Iteration Using Applicators . . . 100
Copying and Referencing Collections . . . 102
Bounded and Unbounded Collections . . . 102

Chapter 9. Element Functions and Key-Type Functions . . . 105
Introduction to Element Functions and Key-Type Functions . . . 105
Using Member Functions . . . 106
Using Separate Functions . . . 107

Using Element Operation Classes 109
Memory Management with Element Operation Classes 113
Functions for Derived Element Classes 113
Using Smart Pointers 115
Overview of Smart Pointers 116
Element Pointers 118
Managed Pointers 120
Automatic Pointers 121
Constructing Smart Pointers 123

Chapter 10. Tailoring a Collection Implementation 127
Introduction 127
Replacing the Default Implementation 127
The Based-On Concept 128
Provided Implementation Variants 128
Features of Provided Implementation Variants 130
Sequences 131
Trees 133
Hash Table 137

Chapter 11. Polymorphism and the Collections 139
Introduction to Polymorphism 139
Using the Abstract Class Hierarchy 139
Adding and Overloading Member Functions 140

Chapter 12. 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 13. Collection Class Library Tutorials 149
Preparing for the Lessons 150
Lesson 1: Defining a Simple Collection of Integers 151
Lesson 2: Adding, Listing, and Removing Elements 153
Lesson 3: Changing the Element Type 159
Lesson 4: Changing the Collection 165
Lesson 5: Changing the Implementation Variant 174
Errors When Compiling or Running the Lessons 176
Other Tutorials 176
Using the Default Classes 176
Advanced Use 177
Source Files for the Tutorials 177

Chapter 14. Solving Problems in the Collection Class Library 179

Cursor Usage 179
Element Functions and Key-Type Functions 180
Key Access Function - How to Return the Key (1) 181
Key Access Function - How to Return the Key (2) 182
Definition of Key-Type Functions 182
Exception Tracing 183
Declaration of Template Arguments and Element Functions (1) 183
Declaration of Template Arguments and Element Functions (2) 184
Declaration of Template Arguments and Element Functions (3) 184
Default Constructor 185

Chapter 15. Compatibility Information 187
Compatible Items 187
Incompatible Items 188

Part 4. The Data Type and Exception Class Library 189

Chapter 16. Data Types and Exceptions 191
Organization of Classes 191
IBase Class 194
IVBase Class 194
String and Buffer Classes 195
DBCS and National Language Support 195

Chapter 17. String Classes 197
Introduction to the String Classes 197
String Buffers 198
Double-Byte Character Set Support 198
Indexing of Strings 198
What You Can Do with Strings 198
Creating and Copying Strings 199
Doing String Input and Output 201
Concatenating Strings 202
Finding Words or Substrings within Strings 202
Replacing, Inserting, and Deleting Substrings 204
Determining String Lengths and Word Counts 205
Extending Strings 205
Converting between Strings and Numeric Data 205
Converting between Strings and Different Base Notations 206
Testing the Characteristics of Strings 207
Formatting Strings 209
Other IString Capabilities 210
IStringTest Class 210

Chapter 18. Exception and Trace Classes 213
Introduction to the Exception Classes 213
Characteristics of the Exception Classes 213
Derivation of the Exception Classes 213

Situations in Which the Exception Classes Are Used . . . 214
Catching Exceptions Thrown by Class Library Functions . . . 215
An Example of the new Operator Throwing an Exception . . . 215
An Example of the Subscript Operator Throwing an Exception . . . 215
Throwing Your Own Exceptions Using the Exception Classes . . . 216
Macros Used with the Exception Classes . . . 217
Why Use the Macros? . . . 219
Using the ITrace Class . . . 220
Using the Trace Macros to Control Trace Output . . . 221
An Example of Using ITrace . . . 222

Chapter 19. 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 20. The IBM Open Class Notification Framework . . . 231
Notifiers and Observers . . . 231
Notification Protocol . . . 233
IBM C++ Notification Class Hierarchy . . . 234

Part 5. The Binary Coded Decimal Class Library for OS/400 . . . 235

Chapter 21. Using the Binary Coded Decimal Classes . . . 237
Introducing _DecimalT Class Template . . . 237
Header File and Constants . . . 237
Constants Defined in bcd.h . . . 238
Constructing _DecimalT Template Class Objects . . . 238
__D Macro . . . 238
_DecimalT Class Template Input and Output . . . 239
Mathematical Operators for _DecimalT Template Class . . . 239
Addition, Subtraction, Multiplication, and Division Operators . . . 239
Relational Operators . . . 240
Equality Operators . . . 240
Conditional Expressions . . . 241

Conversions 241
_DecimalT Template Class to a _DecimalT Template Class 242
A _DecimalT Template Class to an Integer Type 243
A _DecimalT Template Class to a Floating Point Type 243
Size of a _DecimalT Template Class 244
Number of Digits of a _DecimalT Template Class 244
Precision of a _DecimalT Template Class 244
Overflow Behavior 245
_DecimalT Class Template Exceptions 245
Understanding _DecimalT Class Template Run-time Exceptions 245
Using a Class Derived from the _DecErr Class 247
Using Debug Macros 247
Passing a _DecimalT Template Class Object to a Function 249
Passing a Pointer to a _DecimalT Template Class Object 249
Calling Another Program Containing a _DecimalT Template Class 250
Writing a _DecimalT Template Class Constants to a File 251

Appendix A. Using the IBM Open Class Library Source Code 255
Installing the Source Code 255
Building the Source Code 256
Compiling Data Type and Exception Class Library Source Code 256
Compiling Collection Class Library Source Code 256
Binding 257

Appendix B. Class to Implementation File Cross-Reference Table 263

Part 6. Glossary, Bibliography and Index 271

Glossary 273

Bibliography 283
The IBM VisualAge for C++ for AS/400 Library 283
C and C++ Related Publications 283
Other Books You Might Need 283
Non-IBM Publications 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 independent created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact IBM Canada Ltd., Department 071, 1150 Eglinton Avenue East, North York, Ontario M3C 1H7, Canada. Such information may be available, subject to appropriate terms and conditions, including in some cases payment of a fee.

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

Programming Interface Information

This book is intended to help you develop applications that use the C++ class libraries provided with VisualAge for C++ for AS/400. This publication documents General-Use Programming Interface and Associated Guidance Information provided by VisualAge for C++ for AS/400.

General-Use programming interfaces allow the customer to write programs that obtain the services of VisualAge for C++ for AS/400.

© Copyright IBM Corp. 1995, 1996

Trademarks and Service Marks

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

AS/400	C/400
COBOL/400	C Set ++
Common User Access	CUA
IBM	IBMLink
ILE	Integrated Language Environment
Open Class	OS/2
OS/400	PROFS
RPG/400	VisualAge

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

Microsoft and Windows are trademarks or registered trademarks of Microsoft Corporation.

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

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

About This Book

This book gives you guidance on how to use IBM Open Class Library, the comprehensive library of C++ classes that are provided with the VisualAge for C++ for AS/400. IBM Open Class Library consists of the following groups of classes, described individually as "class libraries" in this book:

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

This book also describes the Binary Coded Decimal Class Library for OS/400.

Use this book with the other publications described in the "Bibliography" on page 283.

For information about other AS/400 publications, see either of the following:

- The *Publications Reference* book, SC41-4003, in the AS/400 Softcopy library
- The *AS/400 Information Directory*, a unique, multimedia interface to a searchable database containing descriptions of titles available from IBM or from selected other publishers. The *AS/400 Information Directory* is shipped with your system at no charge.

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

Who Should Use This Book

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

- Manipulate complex numbers (numbers with both a real and an imaginary part)
- Perform input and output to console or disk devices using a typesafe, object-oriented programming approach
- Implement commonly used abstract data types, including sets, maps, sequences, trees, stacks, queues, and sorted or keyed collections
- Manipulate strings with greater ease and flexibility than the standard C++ method of using character pointers and the string functions of the C `string.h` library
- Use date and time information, and apply methods to date and time objects
- Use binary coded decimal objects compatible with the packed decimal data type on the AS/400.

© Copyright IBM Corp. 1995, 1996

Finding Class or Function Descriptions

How to Use This Book

This book is divided into the following chapters and parts:

- **Chapter 1, "Introduction to IBM Open Class Library" on page 1** describes the origins, structure, and uses of each of the class libraries, so that you can decide which libraries or classes to learn about.
- **Part 1, "Complex Mathematics" on page 7** reviews the uses of complex numbers, and describes the `complex` and `c_exception` classes. The `complex` class is used to manipulate complex numbers, and the `c_exception` class is used to handle exceptions resulting from complex number computations.
- **Part 2, "The I/O Stream Library" on page 23** describes the organization of the I/O Stream Class Library, gives reasons for using its classes rather than the I/O interface provided by `stdio.h`, and shows a number of detailed examples of how to do input and output to a console or file, how to format output, how to handle input errors, and so on.
- **Part 3, "The Collection Class Library" on page 71** describes the Collection Class Library, and helps you design applications that use its classes. It discusses instantiating collections, using the Collection Class Library member functions on elements or element keys, tailoring collections for performance, determining the cause of compilation errors resulting from your use of the Collection Classes, and other subjects. This part also contains a set of tutorial lessons that you can use to learn Collection Class Library concepts and techniques.
- **Part 4, "The Data Type and Exception Class Library" on page 189** describes the string, date, time, exception, and other classes that make up the Data Type and Exception Class Library, and shows you how to write programs that use the wide range of string-handling and other features provided by this library.
- **Part 5, "The Binary Coded Decimal Class Library for OS/400" on page 235** describes the `_DecimalT` class template, the `_ConvertDecimal` class and the `_DecErr` class. The `_DecimalT` class template is used to create binary coded decimal objects compatible with the packed decimal data type on the AS/400. The `_ConvertDecimal` class acts as an intermediate class when converting one `_DecimalT` template class to another and converts a large numerical quantity (up to 31 digits) represented in a string literal to the `_DecimalT` template class format. The `_DecErr` class has several derived classes. Objects instantiated from these classes are thrown at runtime to identify various exceptions.

How to Find Class or Function Descriptions

For detailed information on a particular class or member function, see the the *IBM VisualAge for C++ for AS/400 IBM Open Class Library Reference*. If you know what library a class or member function is in, you can turn to the *IBM VisualAge for C++ for AS/400 IBM Open Class Library Reference* section that describes that library. At the beginning of each *IBM VisualAge for C++ for AS/400 IBM Open Class Library Reference* section you will find a list of all classes described in the section, with page references to class descriptions. Each class description in turn includes an alphabetical listing of member functions with page references to individual functions. If you do not

know what section to look in (or what class), you can look up the class or method name in the index.

Classes are organized alphabetically within each class library in the *IBM VisualAge for C++ for AS/400 IBM Open Class Library Reference*, except where classes with similar functionality are placed together. Functions and data members are listed alphabetically at the start of each class chapter, and their descriptions are grouped according to their purpose. If a class has more than one version of a function, all versions are described in one place. For the Collection Classes, all functions of flat collections are described in Chapter 16, “Flat Collection Member Functions” in the *IBM VisualAge for C++ for AS/400 IBM Open Class Library Reference*, because each of these functions is used by many or all of the Collection Classes.

A Note about Examples

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

How to Get Help

There are three kinds of online information available to you while you are using VisualAge for C++ for AS/400:

Online documents
: These are complete documents, like the one you are reading now, presented online. These documents contain detailed information on the different aspects of VisualAge for C++ for AS/400. For your convenience, the online documents are presented in standard format (`.INF` files). See “Getting Help Inside VisualAge for C++ for AS/400” on page xvi for instructions on opening standard format documents from inside VisualAge for C++ for AS/400. See “Getting Help from the Command Line” on page xvi for instructions on opening standard format documents from the command line. For a list of the VisualAge for C++ for AS/400 documents that are available in standard format, see “Online Documents Available in VisualAge for C++ for AS/400” on page xvii.

Contextual help
: Contextual help is available throughout VisualAge for C++ for AS/400. This help tells you all about the elements that you see in the interface, including menus, entry fields, and pushbuttons.

How Do I **help**
: Many of the common tasks that you want to perform with VisualAge for C++ for AS/400 are described in *How Do I* help. The *How Do I* help for a task gives you step-by-step instructions for completing the task. There is overall *How Do I* help for VisualAge for C++ for AS/400, as well as individual task lists for each of its components.

Getting Help Inside VisualAge for C++ for AS/400

All three kinds of help are available directly within the VisualAge for C++ for AS/400 interface:

- To get general contextual help for the component of VisualAge for C++ for AS/400 that you are using, press **F1** anywhere in the window.
- To get contextual help on a particular menu, menu item, or button, highlight the element and press **F1**.
- To get access to all of the help information that is available to you in a particular window, click on **Help** in the menu bar at the top of the window. This menu includes the following selections:
  - **Help Index**, an alphabetical list of all of the help topics that are available from this window
  - **General Help**, overall help for the window
  - **Using Help**, general information about the help facility
  - **How Do I...**, the How Do I help for the component
  - **Product Information**, a dialog that shows the level of VisualAge for C++ for AS/400 being used

  In addition, there are selections that let you open all of online documents that are available in VisualAge for C++ for AS/400.
- To get detailed information, open the **Online Information** notebook in the VisualAge for C++ for AS/400 folder. In this notebook you will find tabs for **Guides**, **References**, and **How Do I** help. Each page in the notebook lists a variety of online documents that describe, in detail, the different aspects of VisualAge for C++ for AS/400. To open a particular online document, select the radio button for the document, and click on the **View** pushbutton.

Getting Help from the Command Line

If you want, you can look at the online documents by issuing the `iview` command. The installation routine stores the online document files in the `\CTTASW\HELP` directory. To view the *Language Reference*, for example, make `C:\CTTASW\HELP` your current directory (substituting the drive where you installed VisualAge for C++ for AS/400 for `C:`) and enter the following command:

```
IVIEW CTTLNG.INF
```

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

```
IVIEW CTTLNG.INF OPERATOR PRECEDENCE
```

Getting Help for a Keyword or Construct

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

Online Documents Available in VisualAge for C++ for AS/400

The following documents are available in standard format:

- VisualAge for C++ for AS/400 C++ Language Reference
- VisualAge for C++ for AS/400 C Library Reference
- VisualAge for C++ for AS/400 IBM Open Class Library Reference
- VisualAge for C++ for AS/400 IBM Open Class Library User's Guide
- VisualAge for C++ for AS/400 C++ User's Guide
- VisualAge for C++ for AS/400 C++ Programming Guide
- IBM Access Class Library for OS/400 Reference
- IBM Access Class Library for Windows Reference
- IBM Access Class Library User's Guide

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.

Note: The IBM Open Classes do not support runtime type information (RTTI). The dynamic_cast and typeid operators will yield unpredictable results when used with Open Class objects. See the Language Reference for more information on these operators; see the Programming Guide for more information on RTTI.

- The *Complex Mathematics Library* provides you with the facilities to manipulate complex numbers and perform standard mathematical operations on them.
- The *I/O Stream Library* gives you the facilities to deal with many varieties of input and output. You can derive classes from I/O Stream classes to customize the input and output facilities for your own particular needs.
- The *Collection Class Library* provides a set of commonly used abstract data types that you can use to build collections. Collections can have properties such as sorted or unsorted, ordered or unordered, unique-element or multiple-element.
- The *Data Type and Exception Classes* let you manipulate string, date, and time information, and let you handle and trace exceptions.

The *Binary Coded Decimal Class Library for OS/400* provides you with the facilities to create binary coded decimal objects compatible with the AS/400 packed decimal data type.

History of IBM Open Class Library

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

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

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

The Binary Coded Decimal Class Library for OS/400 was developed by IBM so that you can create binary coded decimal objects compatible with the packed decimal data type on the AS/400.

© Copyright IBM Corp. 1995, 1996

Class Library Hierarchies

Hierarchies of the Class Libraries

The following figures show the class hierarchy of the class libraries that make up IBM Open Class. Some of these figures are repeated in the parts that describe specific libraries. For a more detailed description of the Collection Class Library hierarchy figure, see “Collection Class Hierarchy” on page 87.

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

No figure is shown for the Binary Coded Decimal Library for OS/400 because the three classes involved `_DecimalT` class template, `_ConvertDecimal` class and `_DecErr` class, are not related by inheritance.

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

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

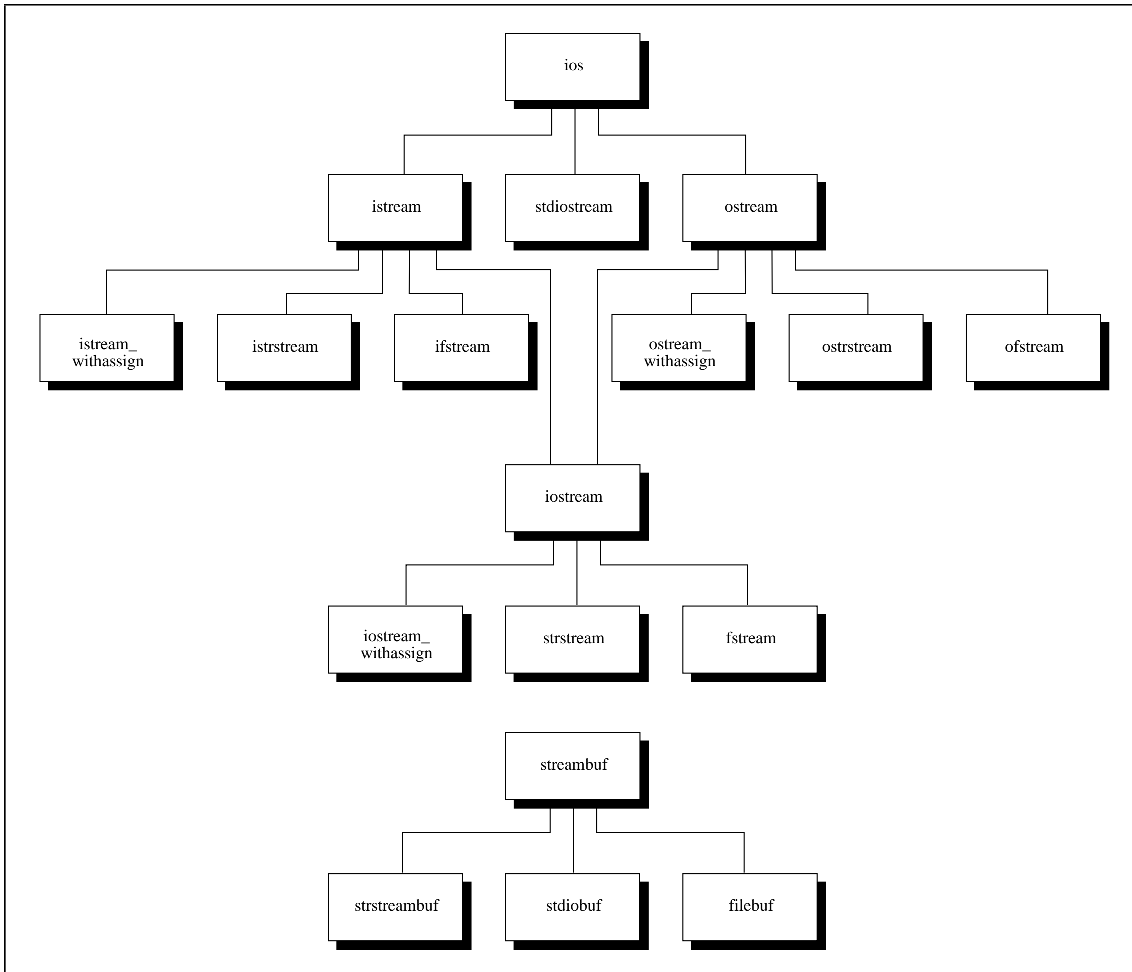


Figure 1. I/O Stream Library Hierarchy

Class Library Hierarchies

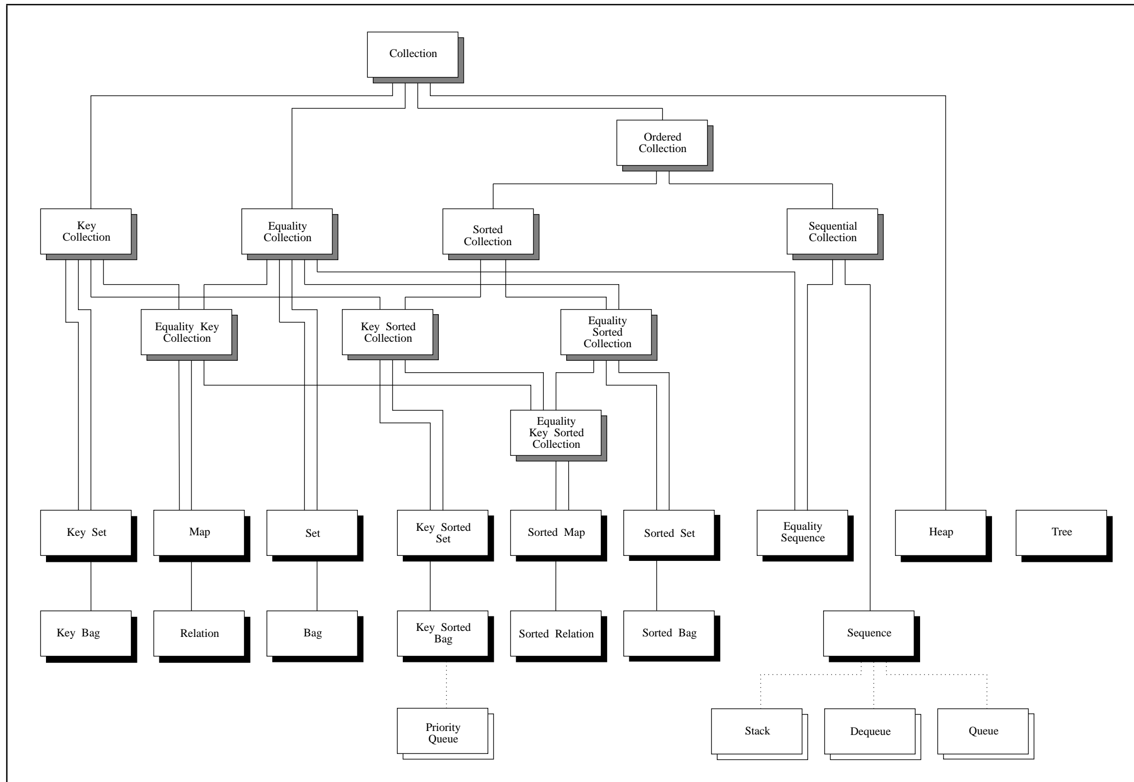


Figure 2. Collection Class Library Hierarchy. Abstract classes have a grey background. Concrete classes have a black background. Restricted access classes have a white background. Dotted lines show a "based-on" relationship, not an actual derivation.

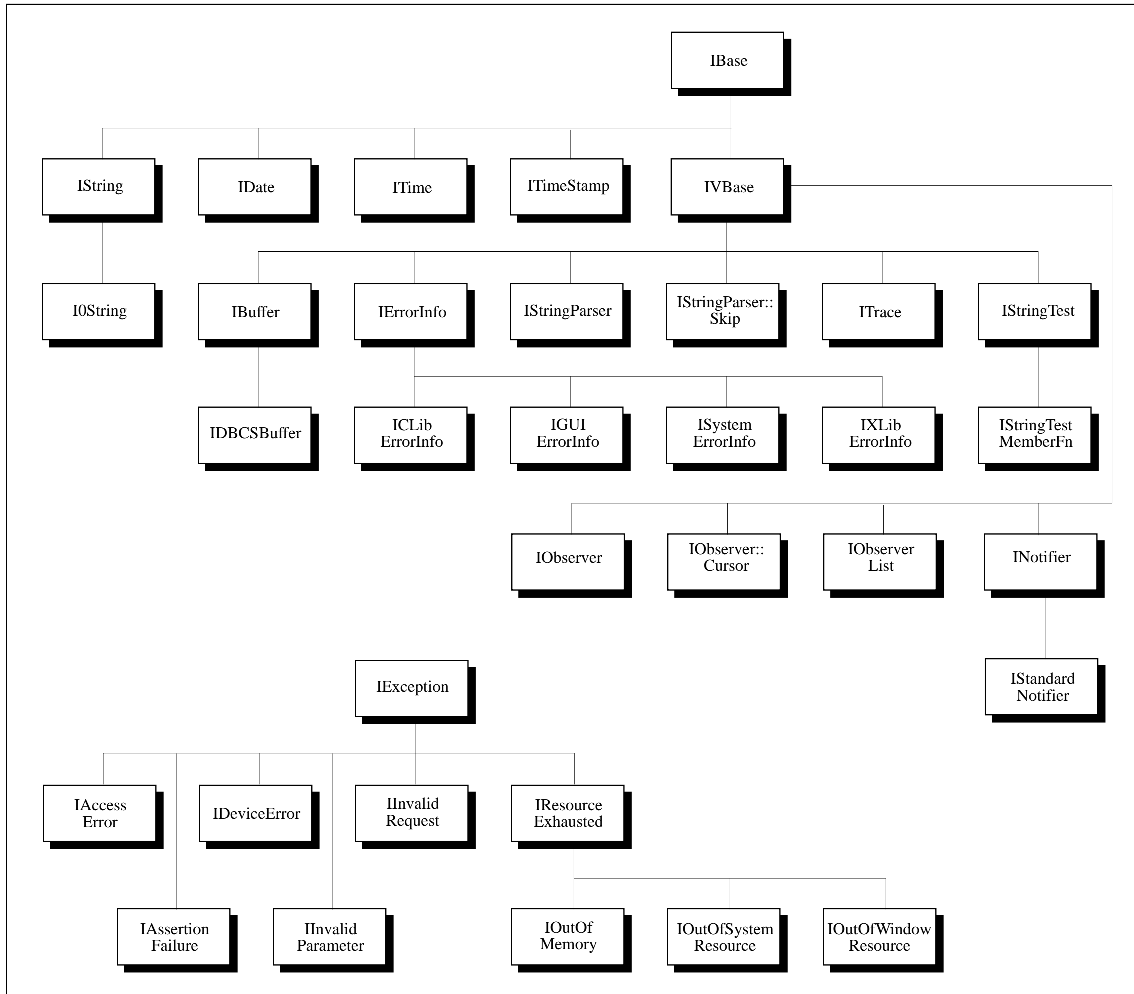


Figure 3. Data Type and Exception Class Hierarchy. Some class names have been split into two lines to fit in their boxes. Note that `IGUIErrorInfo`*,* `IXLibErrorInfo`*, and* `IOutOfSystemResource` *are not supported on VisualAge for C++ for AS/400.*

Including IBM Open Class Library

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

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

Including IBM Open Class Library

the body of your program, you can use a class, function, or operator defined in the header file, as well as derive new classes and overload the functions and operators.

Part 1. Complex Mathematics

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

Chapter 2. Using the Complex Mathematics Classes 9
Review of Complex Numbers 9
Header Files and Constants for complex and c_exception 10
Constructing complex Objects 11
Complex Mathematics Input and Output 11
Mathematical Operators for complex 12
Friend Functions for complex 15
Using the c_exception Class to Handle Complex Mathematics Errors 17
Errors Handled Outside of the Complex Mathematics Library 19
An Example of Using the Complex Mathematics Library 19

© Copyright IBM Corp. 1995, 1996

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 errors created by the functions and operations in the `complex` class. Binding issues, and conflicts between `complex` functions and similarly named functions in the Standard C Runtime Library, are also identified.

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

Review of Complex Numbers

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

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

© Copyright IBM Corp. 1995, 1996

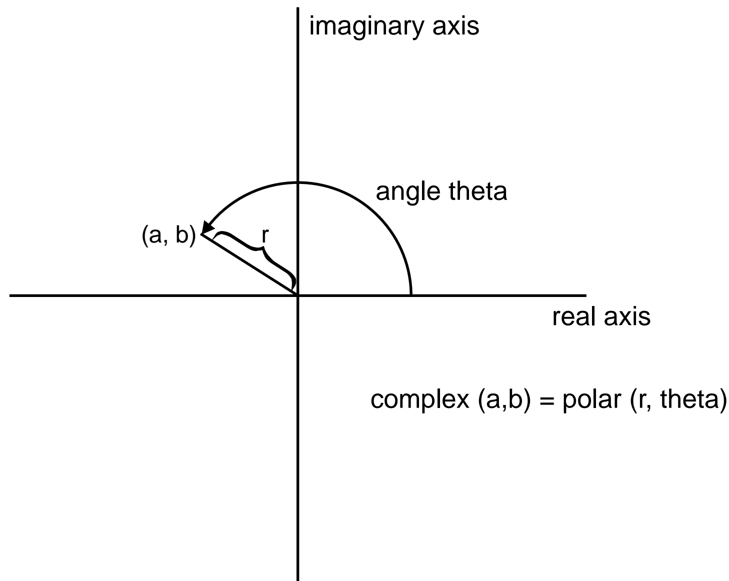


Figure 4. Polar Representation of Complex Number (a,b)

Header Files and Constants for complex and c_exception

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

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

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

Constants Defined in complex.h

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

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

Constant Name	Description
M_E	The constant e
M_LOG2E	The logarithm of e to the base of 2
M_LOG10E	The logarithm of e to the base of 10
M_LN2	The natural logarithm of 2
M_LN10	The natural logarithm of 10
M_PI	π
M_PI_2	π / 2
M_PI_4	π / 4

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

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

Constructing complex Objects

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

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

Complex Mathematics Input and Output

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

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

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

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

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

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

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

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

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

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

Mathematical Operators for complex

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

```
//   Using the complex mathematical operators

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

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

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

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

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

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

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

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

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

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

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

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

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:

Mathematical Operators for complex

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

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

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

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

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

This example produces the following output:

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

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

Assignment Operators Do Not Produce an lvalue

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

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

Friend Functions for complex

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

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

Mathematical Functions for complex

The `complex` class defines four mathematical functions as friend functions of `complex` objects. The functions, described in detail in the *IBM VisualAge for C++ for AS/400 IBM Open Class Library Reference*, are:

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

The following example shows uses of these mathematical functions:

```
// Using the complex mathematical functions

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

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

Friend Functions for complex

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 *IBM VisualAge for C++ for AS/400 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:

```
// 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 *IBM VisualAge for C++ for AS/400 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:

```
// Using the complex conversion functions

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

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

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

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

Using the c_exception Class to Handle Complex Mathematics Errors

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

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

name, arguments, and return value of the function that caused the error, as well as the type of error that has occurred. The data members are:

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

If you do not define your own `complex_error` function, `complex_error` sets the complex return value and the `errno` error number as defined in Table 2 in the *IBM VisualAge for C++ for AS/400 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:

```
//Redefinition of the complex_error function

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

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

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

This example produces the following output:

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

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

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

Compiling a Program that Uses a Customized complex_error Function

If you define your own version of `complex_error`, 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:

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

where ω is a complex number with the value:

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

and σ is a complex number with the value:

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

The following code includes two functions, `get_omega()` and `get_sigma()`, to calculate the values of ω and σ. The user is prompted for the complex value *z* and the value of

Complex Mathematics Library Example

n. After the values of ω and σ have been calculated, the *n* roots of *z* are calculated and printed.

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

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

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

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

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

void main() {
     double n;
     complex input, omega, sigma;
     //
     // prompt the user for a complex number
     //
     cout << "Please enter a complex number: ";
     cin >> input;
     //
     // prompt the user for the value of n
     //
     cout << "What root would you like of this number? ";
     cin >> n;
     //
     // calculate the value of omega
     //
     omega = get_omega(n);
     cout << "Here is omega " << omega << endl;
     //
     // calculate the value of sigma
     //
     sigma = get_sigma(input,n);
     cout << "Here is sigma " << sigma << '\n'
          << "Here are the " << n << " roots of " << input << endl;
     for (int i = 0; i < n ; i++) {
          cout << sigma*(pow(omega,i)) << endl;
     }
}
```

Complex Mathematics Library Example

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)
```

Complex Mathematics Library Example

Part 2. The I/O Stream Library

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

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
The I/O Stream Class Hierarchy 26
The I/O Stream Header Files 28
Predefined Streams 29
Anonymous Streams 30
Stream Buffers 31
Format State Flags 33

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 endl and flush 40
Parsing Multiple Inputs 42
Opening a File for Input and Reading from the File 43
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 49
Defining an Output Operator for a Class Type 51
Correcting Input Stream Errors 53
Changing the Formatting of Stream Output 55
Defining Your Own Format State Flags 61
Using the strstream 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

© Copyright IBM Corp. 1995, 1996

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

The I/O Stream Classes predefine streams for standard input, standard output, and standard error. See “Predefined Streams” on page 29 for more details on the predefined streams. If you want to open your own streams for input or output, you must create an object of an appropriate I/O Streams class. The `iostream` constructor

© Copyright IBM Corp. 1995, 1996

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:

```
cout << 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 49 and “Defining an Output Operator for a Class Type” on page 51 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 31 for information on how and why to use stream buffers. `streambuf` is the base class for the following classes:
  - `strstreambuf`
  - `stdiobuf`
  - `filebuf`
- The `ios` class maintains formatting and error state information for streams. Streams are implemented as objects of the following classes that are derived from `ios`:
  - `stdiostream`
  - `istream`
  - `ostream`

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

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

- ifstream
- istream_withassign
- iostream

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

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

I/O Stream Header Files

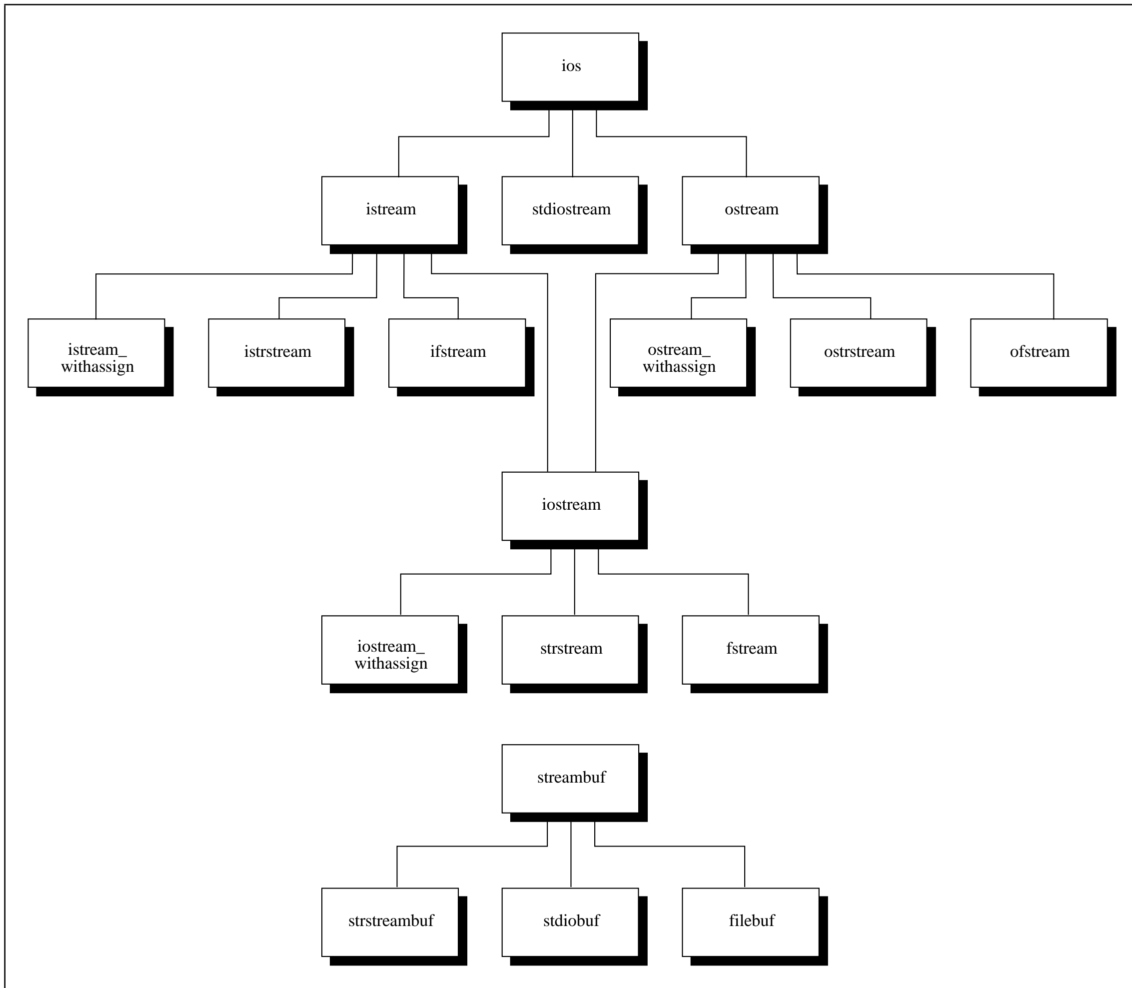


Figure 5. I/O Stream Class Hierarchy

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

The I/O Stream Header Files

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

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

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

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

Predefined Streams

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

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

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

The predefined streams `cin`, `cerr`, and `clog` are *tied* to `cout`. As a result, if you use `cin`, `cerr`, or `clog`, `cout` is *flushed.* That is, the contents of `cout` are sent to their ultimate consumer. See “tie” in the *IBM VisualAge for C++ for AS/400 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 AT&T C++ Language System Release 2.1, the VisualAge for C++ for AS/400 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.

```
// 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::o ut);         // define and open the file
   File2 << c << endl;                          // write to the file
   File2.close();                               // close the file
   File2.open("file2.abc",ios::in);             // reopen for input
   File2 >> c;                                  // read from the file
   cout << c << endl;                           // show what was in the file
   }
```

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

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

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

Stream Buffers

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

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

What Does a Stream Buffer Do?

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

Why Use a Stream Buffer?

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

Consider, for example, an application that is reading data from the ultimate producer. If there is no buffer, the application has to make a system call for each character that is

read. However, if the application uses a stream buffer, system calls will only be made when the buffer is empty. Each system call will read enough characters from the ultimate producer (if they are available) to fill the buffer again.

How Is a Stream Buffer Implemented?

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

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

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

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

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

Figure 6 on page 33 shows how the pointers returned by these functions delineate the stream buffer.

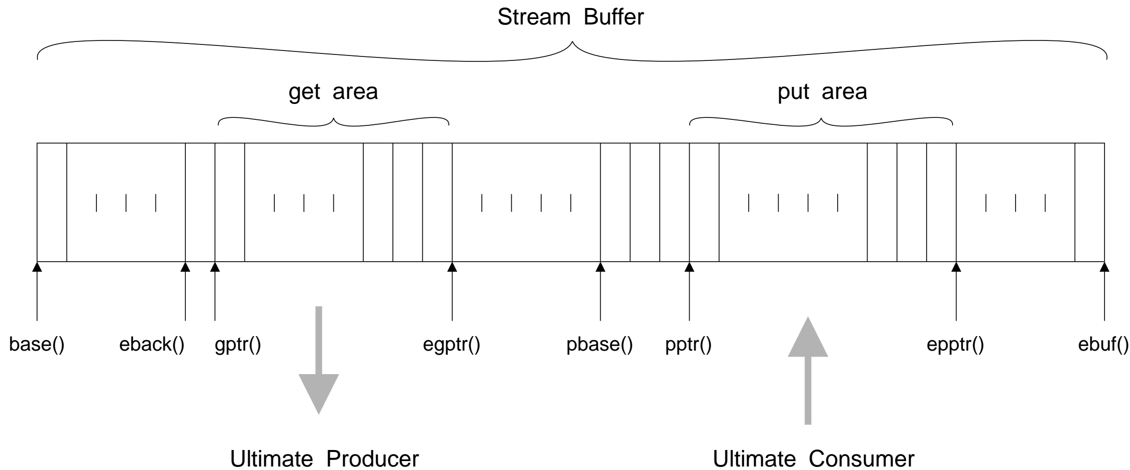


Figure 6. The Structure of Stream Buffers

Format State Flags

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

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

For examples of how to use these format state flags, see “Changing the Formatting of Stream Output” on page 55. For descriptions of individual format state flags, see “Format State Flags” in the *IBM VisualAge for C++ for AS/400 IBM Open Class Library Reference*.

Format State Flags

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 49 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>>".
```

© Copyright IBM Corp. 1995, 1996

Receiving Input

Multiple Variables in an Input Statement

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

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

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

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

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

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

String Input

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

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

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

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

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

White Space in String Input

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

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

#include <iostream.h>

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

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

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

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

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

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

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

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

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

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

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

of input is read by the second getline() call, and the program ends before any further input can be read.

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

Incorrect Input and the Error State of the Input Stream

When your program requests input through the input operator and the input provided is incorrect or of the wrong type, the error state may be set in the input stream and further input from that input stream may fail. One runtime symptom of such a failure is that your program's prompts for further input display without pausing for the input. See "Correcting Input Stream Errors" on page 53 for details on how to detect and correct input stream errors.

Using Input Streams Other Than cin

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

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

Displaying Output on Standard Output or Standard Error

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

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

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

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

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

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

An output operator must exist for any type being output. The I/O Stream Library defines output operators for all C++ built-in types. For types you define yourself, you need to provide your own output operators. See “Defining an Output Operator for a Class Type” on page 51 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;
```

Flushing Output Streams

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 *IBM VisualAge for C++ for AS/400 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."
- The program terminates.

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

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

Placing endl or flush in an Output Stream

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

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

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

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

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

is not as efficient as:

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

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

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

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

Parsing Multiple Inputs

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

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

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

Input	Output	Remarks
123<\n>		No output. a has been assigned the value 123, but the program is still waiting on input for b and c.
1<\n> ,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 int a up to the first character that is not acceptable input for an integer (the comma). Characters are read into int b where input for a left off (the comma). Because a comma is not one of the allowable characters for integer input, ios::failbit is set, and all further input fails until ios::failbit is cleared. See "Correcting Input Stream Errors" on page 53 for details on how to clear an input stream.
1.2 2.3<\n> 3.4<\n>	a: <1> b: <-559038737> c: <-559038737>	As with the previous example, characters are read into a until the first character is encountered that is not acceptable input for an integer (in this case, the period). The next input of an int causes ios::failbit to be set, and so both it and the third input result in errors.

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

Opening a File for Input and Reading from the File

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

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

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:

File Input

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

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

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

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

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

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

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

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

```
barbers often shave
```

the file contains the following after the program is run:

```
barbers often shave
barbers
```

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

Reading Input from a File

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

stdin.C

```
#include <iostream.h>

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

filein.C

```
#include <fstream.h>

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

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

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

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

For more information on defining your own input operators, see "Defining an Input Operator for a Class Type" on page 49.

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

Opening a File for Output and Writing to the File

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

File Output

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 *IBM VisualAge for C++ for AS/400 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.

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 51.

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 `strstream` classes to accept input from and to send output to character arrays (strings).

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

Associating a File with a Standard Input or Output Stream

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

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

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

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

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

© Copyright IBM Corp. 1995, 1996

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

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

Using filebuf Functions to Move Through a File

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

the output of the example program is:

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

Defining an Input Operator for a Class Type

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

myclass.h

Defining Your Own Input Operator

```
#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
};
```

Defining Your Own Output Operator

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

The output operator must have the following characteristics:

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

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

Class Output Operators and the Format State

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

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

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

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

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


ostream& operator << (ostream& aStream, Coord_3D coord) {
   int startingWidth=aStream.width();
   aStream << coord.X
```

```
#ifndef NOSETW
           << setw(startingWidth)  // set width again
#endif
           << coord.Y
#ifndef NOSETW
           << setw(startingWidth)  // set width again
#endif
           << coord.Z;
   return aStream;
   }

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

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

```
      38.16221773.5917293.1
38.16221773.5917293.1
```

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

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

See "Changing the Formatting of Stream Output" on page 55 for more information on the format state and how to change it, within output operators and in client code.

Correcting Input Stream Errors

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

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

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

Correcting Input Stream Errors

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

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

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

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

All of these methods are described in Chapter 6, “ios Class” on page 31 in the *Open Class Library Reference*.

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

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

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

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

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

When you define an input operator of class type, you can build error-checking code into your definition. If you do so, you do not have to check for error-causing input every

time you use the input operator for objects of your class type. Consider the class definition for the PhoneNumber data type shown in "myclass.h" on page 49, 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 51 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.

Formatting Your Output

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

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

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

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

1. The original settings of the format state flags are determined, using flags(). These settings are saved in the variable originalFlags.

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

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

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

#include <iostream.h>

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

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

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

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

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

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

This example produces the following output:

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

Note:

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

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

Changing the Notation of Floating-Point Values

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

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

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

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

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

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

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

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

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

#include <iostream.h>

void main() {
   double a=3.14159265358979323846;
   double b;
   long originalFlags=cout.flags();
```

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

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

The output from this program is:

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

Changing the Base of Integral Values

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

```
//Showing the base of integer values

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

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

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

Setting the Width and Justification of Output Fields

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

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

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

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

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

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

Defining Your Own Format State Flags

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

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

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

```
// Defining your own format flags

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

long InFileFormat=0;
long OutFileFormat=0;

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

ostream& operator<<(ostream &os, CustRecord &cust) {
   os << cust.Number << '\n'
      << cust.Name   << '\n'
      << cust.Phone  << '\n'
      << cust.Street << '\n'
      << cust.City   << '\n';
```

Defining Your Own Format State Flags

```
   if (os.flags() & OutFileFormat) // New file format
      os << cust.Country << '\n'
         << cust.PostCode << endl;
   else                                  // Old file format
      os << cust.PostCode << '\n'
         << cust.Country << endl;
   return os;
   }

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

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

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

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

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

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

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

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

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

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

Using the strstream Classes for String Manipulation

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

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

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

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

String Manipulation Using strstream

- Your application needs to parse the environment string or another string already in memory, as if that string were formatted input.

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

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

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

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

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

This program produces the following output:

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

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

Simple Manipulators and Parameterized Manipulators

There are two kinds of manipulators:

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

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

```
// Using simple and parameterized manipulators

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

© Copyright IBM Corp. 1995, 1996

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 69, 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
`istream`	`IMANIP(`*tp*`)` or `IAPP(`*tp*`)`
`ostream`	`OMANIP(`*tp*`)` or `OAPP(`*tp*`)`
`iostream`	`IOMANIP(`*tp*`)` or `IOAPP(`*tp*`)`
The `ios` part of `istream` objects or `ostream` objects	`SMANIP(`*tp*`)` or `SAPP(`*tp*`)`

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

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

4. If you chose one of the APP classes in step 2, define the manipulator as shown in "Example of Defining an APP Parameterized Manipulator." If you chose one of the MANIP classes in step 2, define the manipulator as shown in "Example of Defining a MANIP Parameterized Manipulator" on page 69. 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.

```
//   Creating and using parameterized manipulators

#include <iomanip.h>

// declare class

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

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

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

IOMANIPdeclare(my_class);

// define a manipulator for the class my_class

OAPP(my_class) pre_print=produce_prefix;

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

This program produces the following output:

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

Example of Defining a MANIP Parameterized Manipulator

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

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

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

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

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

IOMANIPdeclare(my_class);

// define a manipulator for the class my_class

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

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

This example produces the same output as the previous example.

Examples of Nonassociative Parameterized Manipulators

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

```
// Nonassociative parameterized manipulators

#include <iomanip.h>

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

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

iostream_withassign ioswa;

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

Part 3. The Collection Class Library

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 82
Trees 83
Auxiliary Classes 84
The Overall Implementation Structure 84
Binding to the Collection Classes 89

Chapter 8. Instantiating and Using the Collection Classes 91
Instantiation and Object Definition 91
Adding, Removing, and Replacing Elements 92
Cursors 95
Iterating over Collections 98
Copying and Referencing Collections 102
Bounded and Unbounded Collections 102

Chapter 9. Element Functions and Key-Type Functions 105
Introduction to Element Functions and Key-Type Functions 105
Using Member Functions 106
Using Separate Functions 107
Using Element Operation Classes 109
Functions for Derived Element Classes 113
Using Smart Pointers 115

Chapter 10. Tailoring a Collection Implementation 127
Introduction 127
Replacing the Default Implementation 127
The Based-On Concept 128
Provided Implementation Variants 128
Features of Provided Implementation Variants 130

Chapter 11. Polymorphism and the Collections 139
Introduction to Polymorphism 139
Using the Abstract Class Hierarchy 139
Adding and Overloading Member Functions 140

Chapter 12. Exception Handling 143
Introduction to Exception Handling 143
Precondition and Defined Behavior 144
Levels of Exception Checking 145
List of Exceptions 145

© Copyright IBM Corp. 1995, 1996

The Hierarchy of Exceptions 147

Chapter 13. Collection Class Library Tutorials 149
Preparing for the Lessons 150
Lesson 1: Defining a Simple Collection of Integers 151
Lesson 2: Adding, Listing, and Removing Elements 153
Lesson 3: Changing the Element Type 159
Lesson 4: Changing the Collection 165
Lesson 5: Changing the Implementation Variant 174
Errors When Compiling or Running the Lessons 176
Other Tutorials 176

Chapter 14. Solving Problems in the Collection Class Library 179
Cursor Usage 179
Element Functions and Key-Type Functions 180
Key Access Function - How to Return the Key (1) 181
Key Access Function - How to Return the Key (2) 182
Definition of Key-Type Functions 182
Exception Tracing 183
Declaration of Template Arguments and Element Functions (1) 183
Declaration of Template Arguments and Element Functions (2) 184
Declaration of Template Arguments and Element Functions (3) 184
Default Constructor 185

Chapter 15. Compatibility Information 187
Compatible Items 187
Incompatible Items 188

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 *IBM VisualAge for C++ for AS/400 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

© Copyright IBM Corp. 1995, 1996

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.

Concrete Classes

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.

Concrete Classes

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 83 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 95 for more details on cursor classes. See "Iteration Using Applicators" on page 100 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 115 and "Managed Pointers" on page 120 for more details on pointer classes.

Flat Collections

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

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

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

Equality for elements
: Whether equality is defined for the element.

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

Figure 7 on page 79 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.

<table>
<tr><th colspan="2" rowspan="3"></th><th colspan="2" rowspan="2">Unordered</th><th colspan="3">Ordered</th></tr>
<tr><th colspan="2">Sorted</th><th>Sequential</th></tr>
<tr><th>Unique</th><th>Multiple</th><th>Unique</th><th>Multiple</th><th>Multiple</th></tr>
<tr><th rowspan="2">Key (Key equality must be defined)</th><th>Element Equality</th><td>Map</td><td>Relation</td><td>Sorted map</td><td>Sorted relation</td><td>N/A</td></tr>
<tr><th>No Element Equality</th><td>Key set</td><td>Key bag</td><td>Key sorted set</td><td>Key sorted bag</td><td>N/A</td></tr>
<tr><th rowspan="2">No Key</th><th>Element Equality</th><td>Set</td><td>Bag</td><td>Sorted set</td><td>Sorted bag</td><td>Equality sequence</td></tr>
<tr><th>No Element Equality</th><td>N/A</td><td>Heap</td><td>N/A</td><td>N/A</td><td>Sequence</td></tr>
</table>

Figure 7. 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.

Flat Collections

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 16, “Flat Collection Member Functions” in the *IBM VisualAge for C++ for AS/400 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 80, this means defining job equality as equality of the job ID.) Use this alternative only when you are sure that keys are unique. When you use this alternative, you can locate an element only with the key (using `locateElementWithKey(`*key*`)` instead of `locate(`*element*`)`. Locating elements by key improves performance, particularly if the complete element is large or difficult to construct in comparison to the key alone. Consider the two alternatives in the following example:

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

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

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

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

Uniqueness of Entries

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

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

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

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

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

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 16, “Flat Collection Member Functions” in the *IBM VisualAge for C++ for AS/400 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.
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 *IBM VisualAge for C++ for AS/400 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 2 on page 83 to select the appropriate flat collection with restricted access for a given set of properties.

Table 2. Properties for Collections with Restricted Access

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

Trees

Trees can be described either as structures where the elements have a hierarchy or as a special form of recursive structure. Recursively a tree can be described as a node (parent) with pointers to other nodes (children). Every node has a fixed number of pointers, which are set to null at initialization time. Insertion of a new node involves setting a pointer in the parent so that it points to the inserted child. Figure 8 illustrates the structure of an n-ary tree.

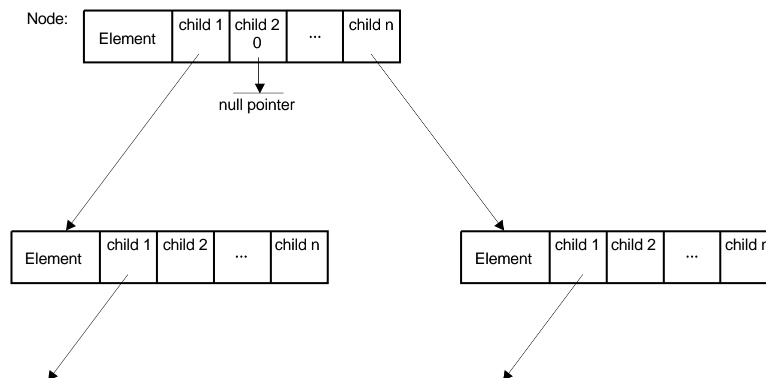


Figure 8. The Structure of N-ary Trees

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 95 and “Iteration Using Applicators” on page 100.

You can use the *smart pointer* classes to manage objects; they enable automatic storage management. “Using Smart Pointers” on page 115 and “Managed Pointers” on page 120 explain the concepts and usage in detail.

The Overall Implementation Structure

To achieve maximum runtime efficiency and ease of use, the Collection Class Library combines the common features of object-oriented techniques, such as class hierarchies, polymorphism and late binding, with an efficient class structure that uses advanced optimization techniques. This section gives a brief overview of the Collection Class structure that is shown in Figure 9 on page 86. 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 *IBM VisualAge for C++ for AS/400 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 implentation 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 9 on page 86 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 88.

Implementation Structure

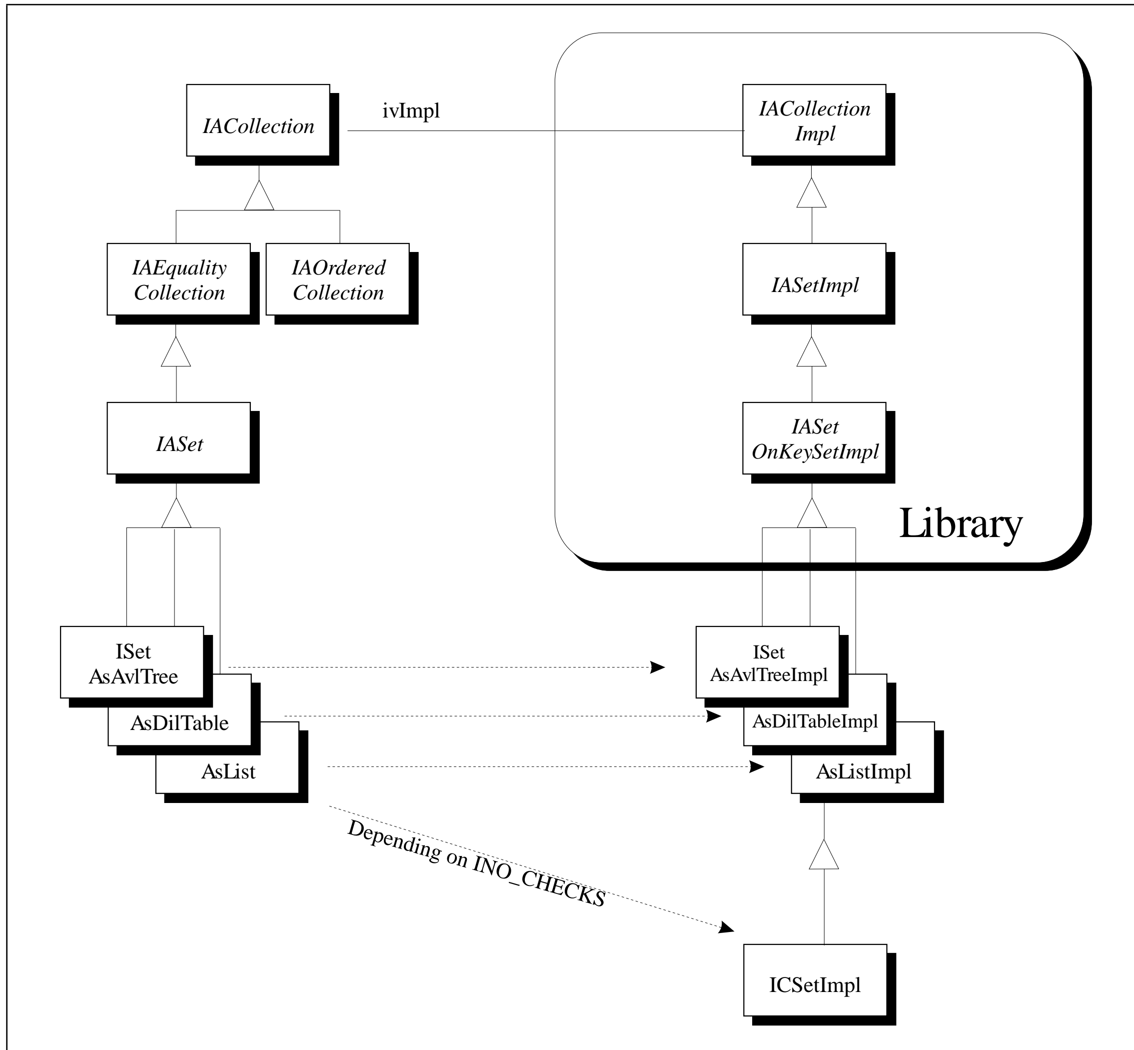


Figure 9. Overall Structure of Collection Classes

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 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 127.

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 109 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 10 on page 88. 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 139 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 a collection of integers. However, suppose an application were to use all of the following:

Implementation Structure

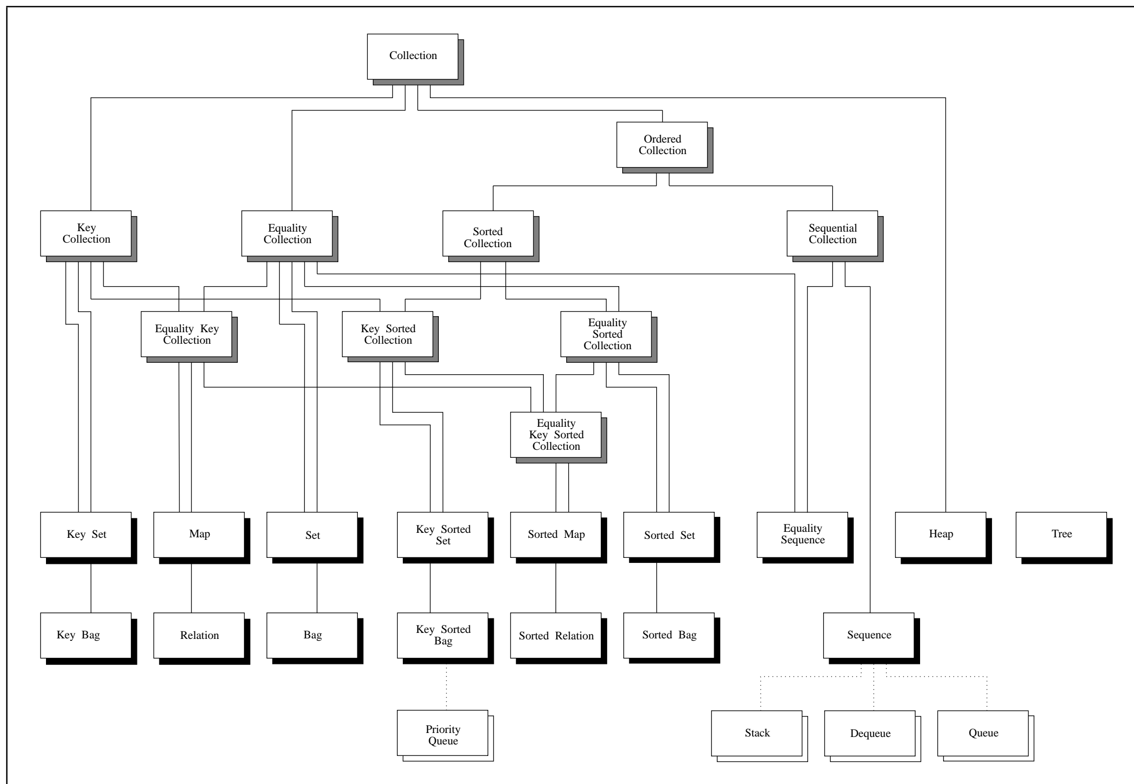


Figure 10. The Collection Class Hierarchy. Abstract classes have a grey background. Concrete classes have a black background. Restricted access classes have a white background. Dotted lines show a "based-on" relationship, not an actual derivation.

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

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

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

Class Template Naming Conventions

All class templates begin with an uppercase `I`. Table 3 on page 89 shows the naming conventions used to distinguish between different types of class templates, given a

default class template of ISet. Underscored letters in each class template name are those that indicate the stated convention:

Class name	Meaning of letters
ISet	Default class template.
ISetImpl	Typeless implementation class.
ICSetImpl	Typeless implementation class that implements additional checks.
IGSet	Default generic class template. The element operations class can be specified as template argument.
ISetAsAvlTree ISetAsBstTree ISetAsList ISetAsTable ISetAsDilTable ISetAsHshTable	Variant class templates.
IASet	Abstract class template.

Table 3. Class Template Naming Conventions

Binding to the Collection Classes

The Collection Class Library resides in the service program QYPPCCL in library QSYS. If you use collection classes in your code, you do not have to bind to this service program; binding is automatic for service programs that reside in QSYS.

Binding to the Collection Class Library

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 <istring.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";
}
```

© Copyright IBM Corp. 1995, 1996

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

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

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 127 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 11 on page 93 shows the result of adding a series of four elements to a map, a relation, a key set, and a key bag. The first row shows what each collection looks like after the element <a,1> has been added to each collection. Each following row shows what the collections look like after the element in the leftmost column is added to each.

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

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

Figure 11. 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 95. There is an important difference between element *values* and lement *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 `IBoolean` value. The `IBoolean` 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;

IBoolean 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 99 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 96. The `replaceAt()` function checks whether the replacing element has the same positioning property as the replaced element. (See Chapter 12, "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);
```

Cursors

```
    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 99.)
- 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 92 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 16, “Flat Collection Member Functions” in the *IBM VisualAge for C++ for AS/400 IBM Open Class Library Reference*), that takes a predicate function as argument. See “Removing Elements” on page 93 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 95). 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 92 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.

Iteration

The function returns an `IBoolean` 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 apllicator 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.

```
 //    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:

```
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`.

Bounded and Unbounded Collections

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

© Copyright IBM Corp. 1995, 1996

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 *IBM VisualAge for C++ for AS/400 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 13, “Collection Class Library Tutorials” on page 149) and in the coding examples in the *IBM VisualAge for C++ for AS/400 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 the bibliography.

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 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 <istring.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");
    forCursor(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 *IBM VisualAge for C++ for AS/400 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;
     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 IGKeySet <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: ";
   forCursor(myCursor2) {
      cout<<PhoneBook.elementAt(myCursor2);
   }

   cout << "\n\nPhoneBook after removing an element: ";
   PhoneBook.removeElementWithKey("714-50706");
   forCursor(myCursor2) {
      cout<<PhoneBook.elementAt(myCursor2);
   }

   cout << "\n\nBusiness before removing an element: ";
   forCursor(myCursor1) {
      cout<<Business.elementAt(myCursor1);
   }

   cout << "\n\nBusiness after removing an element: ";
   Business.removeElementWithKey("Peter Black");
   forCursor(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 *IBM VisualAge for C++ for AS/400 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`.

Functions for Derived Element Classes

The classes IElemPointer, IMngElemPointer, and IAutoElemPointer (see “Managed Pointers” on page 120) 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. 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);

   forCursor (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`

Using Smart Pointers

- `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 118 for more information on the element pointer types.

Using Smart Pointers

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 121 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 120 for more information on managed pointers.

To exploit the advantage of memory management, you can use non-element pointers (for example, `IMngPointer`) instead of standard C++ pointers without storing the pointers in a collection.

Automatic storage management is particularly useful when functions return pointers or references to objects that they have created (dynamically allocated), and the last user of the object is responsible for cleaning up.

The following features of Collection Classes pointer types give you the choices shown in the table below. Standard C++ pointers are included for comparison.

- Element functions performed on referenced elements
- Element functions performed on pointers
- Automatic storage management

Using Smart Pointers

	Destruction of Pointed Objects		
	Not managed	**When out-of-scope**	**Reference counted**
Collections call element operations on pointer	Standard C++ pointer	IAutoPointer	IMngPointer
Collections call element operations on referenced object	IElemPointer	IAutoElemPointer	IMngElemPointer

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:

```
  //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);

   forCursor (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

Using Smart Pointers

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);

    forCursor (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);

   forCursor (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).

   forCursor (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:

```
IString *s = new IString("...");
IMngPointer < IString > p1 (s, IINIT); // OK
IMngPointer < IString > p2 (s, IINIT); // NO!
// Do not use s a second time, because the compiler may try to
// delete the IString object referred to by s, p1, and p2 twice.
```

Using Smart Pointers

> 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.)

Using Smart Pointers

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 84, 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

© Copyright IBM Corp. 1995, 1996

particular implementation variants. Required functions for a collection's default implementation or an implementation variant are listed in the collection's chapter in the *IBM VisualAge for C++ for AS/400 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 *IBM VisualAge for C++ for AS/400 IBM Open Class Library Reference*. See Figure 12 on page 129 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 *IBM VisualAge for C++ for AS/400 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 12 on page 129 lists the basic and based-on implementations provided by the Collection Classes. The upper left corner of each cell contains the name of the (abstract) collection class; basic implementations are written in smaller letters in bold face, while based-on implementations are described by arrows starting from the class that they implement and ending in the (abstract) class on which they are based. An implementation choice for a given class must use either a basic implementation for this class or follow a based-on implementation path that ultimately leads to a basic implementation.

Take the example of the `Bag` abstraction. The `Bag` is not implemented directly. (You can tell this from the figure because no implementation variant name appears in bold in the box containing `Bag`.) To determine the possible implementation variants for `Bag`, follow the arrows out of the `Bag` box:

- One arrow leads to the KeySet box. The KeySet box contains an implementation variant, **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 Bag can therefore be implemented using any of the six implementation variants cited in bold face above.

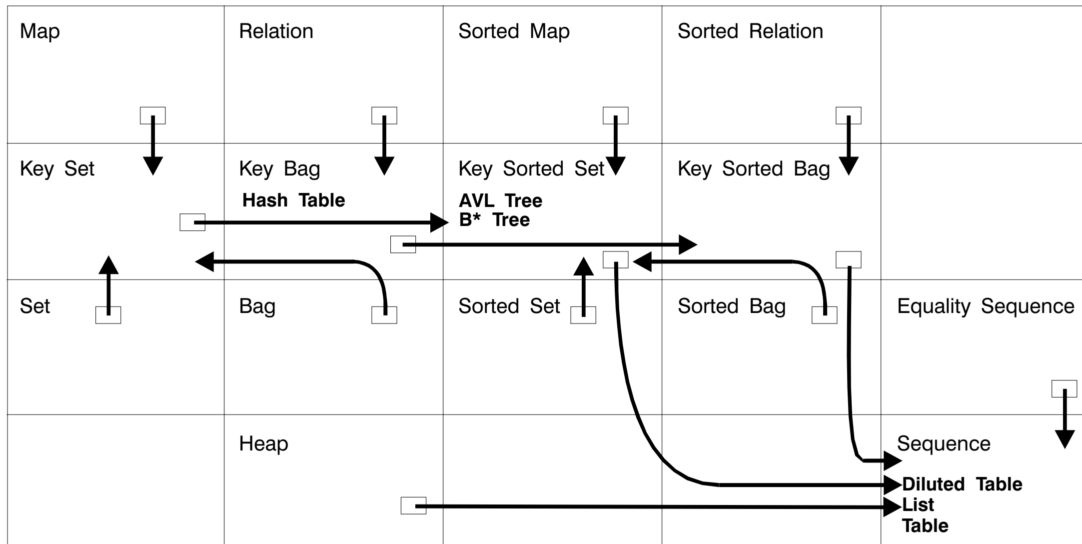


Figure 12. Possible Implementation Paths

Provided Implementation Variants

Table 4. 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 13. 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 128, and are listed in the “Class Implementation Variants” section of each collection chapter in the *IBM VisualAge for C++ for AS/400 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 22, “Key Set” in the *IBM VisualAge for C++ for AS/400 IBM Open Class Library Reference*, the section “Variants and Header Files” on page 153 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 14 shows a list implementation variant.

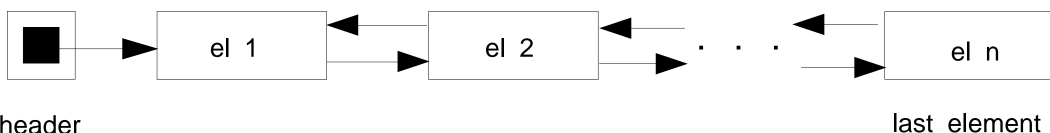


Figure 14. List Implementation Variant

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 15 shows a table implementation variant.

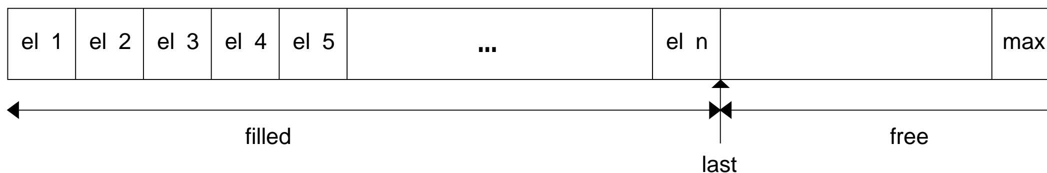


Figure 15. Table Implementation Variant

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.)

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 16 on page 133 shows a diluted table implementation variant.

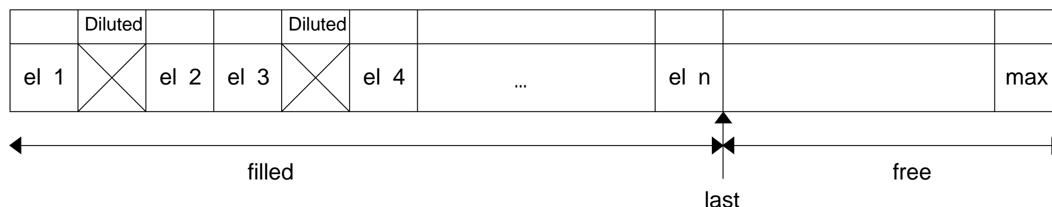


Figure 16. Diluted Table Implementation Variant

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 17 on page 134 has elements added in the following sequence: *8 - 10 - 5 - 1 - 9 - 6 - 11*. A search for element 9 begins at the root node (element 8). Assuming that the element value defines the ordering relation, the search would take the right node from element 8 (because 9 is greater than 8) and would arrive at element 10. The search would take the left node from element 10 (because 9 is smaller than 10) and would arrive at element 9, the desired element.

Provided Implementation Variants

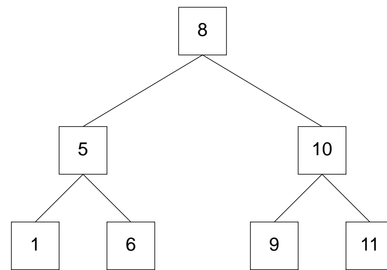


Figure 17. Binary Search Tree

One drawback of a binary search tree is that the tree can easily become unbalanced. Figure 18 shows how unbalanced the tree becomes when the elements 12 through 15 are added.

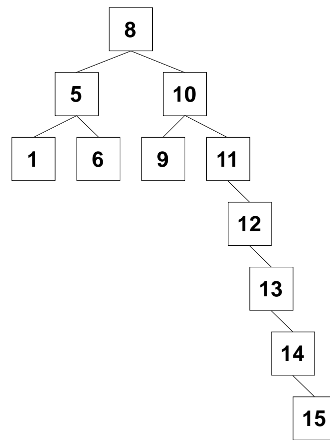


Figure 18. Unbalanced Binary Search Tree

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 19 on page 135 shows how the unbalanced tree shown earlier would look after the AVL property is restored.

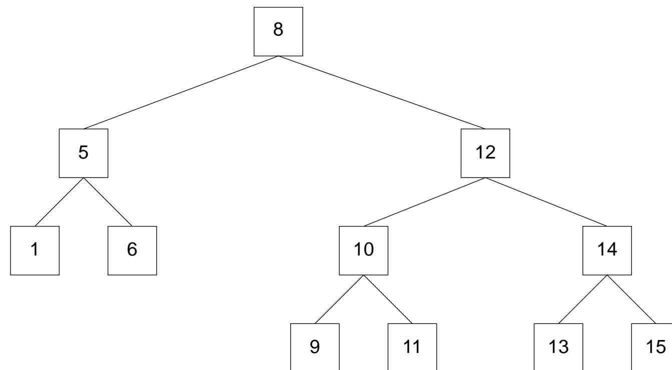


Figure 19. AVL Tree

AVL trees are useful for collections containing a large number of small elements. An AVL tree implementation is even suitable for adding and deleting, because the performance overhead for the rebalancing that sometimes occurs when an element is added or deleted is still less expensive than searching through the elements of a sequence to find the position at which to add or delete an element.

If you use a set collection and do not choose an implementation variant, you are automatically using an AVL tree. If you use a set and are not aware that the set is implemented as an AVL tree, you may be surprised that a set requires an ordering relation, when a set is an *unordered* collection, as shown in Figure 7 on page 79. 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 20 shows a B* tree with up to five children per node.

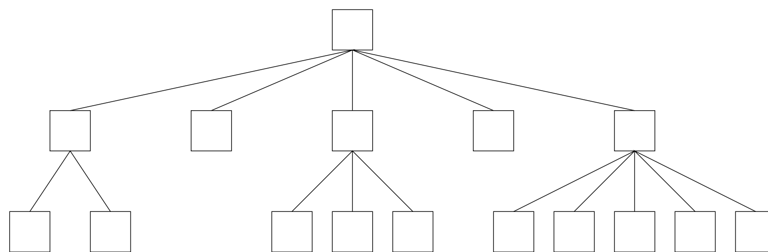


Figure 20. A B tree*

Provided Implementation Variants

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 21, 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.

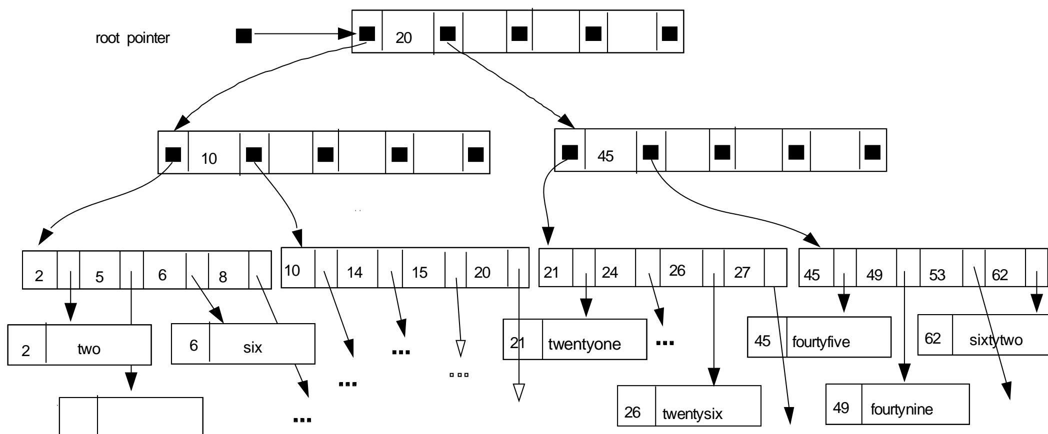


Figure 21. B Tree Implementation Variant*

Hash Table

Hashing is another important and widely used technique to implement collections. Conceptually, hashing involves calculating an index from the key or other parts of an element, and then using that index to look for matches in a hash table. The function that calculates the index is called a *hash function*.

A hash table implementation variant is suitable for nearly all applications with a balanced mix of operations. Such an implementation is quick for retrieving elements. It can also add and delete elements quickly, because, unlike an AVL tree, it does not need to be rebalanced. The efficiency of a hash-table implementation is largely dependent on how efficiently you implement the hash function.

You cannot use a hash-table implementation variant when you require your elements to appear in main storage in sorted order (where elements earlier in the sorting order have lower addresses than elements later in the sorting order). On the other hand, you must use a hash table if you have a complex key (one composed, for example, of several attributes of an element), and either you cannot find a reasonable way to compare keys, or the comparison would be expensive.

For collections that do not provide access by key, but that support a hash-table implementation variant, the complete element is used as the input to the hash function.

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 22 shows a hash table implementation variant.

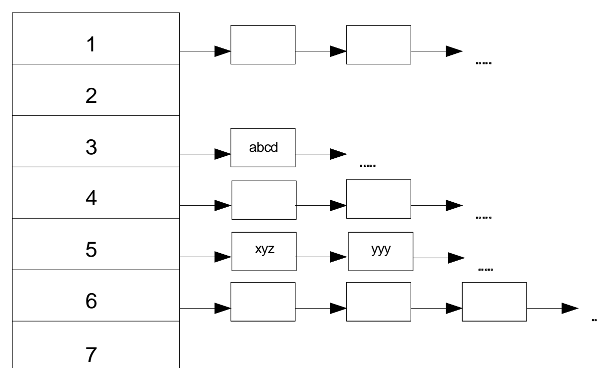


Figure 22. Hash Table Implementation Variant

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 10 on page 88.

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 140.

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

© Copyright IBM Corp. 1995, 1996

that is defined for the given collection. The key set `running` can be used as argument to the universal printer.

```
class JobPrinter {
public:
     print (IACollection <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.

Polymorphism and the Collections

Chapter 12. 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.

© Copyright IBM Corp. 1995, 1996

Exceptions Caused by System Failures and Restrictions

System failures and restrictions are different from precondition violations. You cannot usually anticipate them, and you have no opportunity to verify that such situations, for example storage overflow, will not occur. These exceptions need to be checked for, and an exception should be thrown if they occur.

Precondition and Defined Behavior

Exceptions are not generally used to change the flow of control of a program under normal circumstances. An example of using exceptions under normal circumstances is a function that iterates through a collection, and exits from the iteration by checking for the exception that is thrown when an invalid cursor is used to access elements. When the iteration is complete, the cursor will no longer be valid, and this exception will be thrown. This is not a good programming practice. A function should explicitly test for the cursor being valid. To make this possible, a function must efficiently test this condition (`isValid()`, for the cursor example).

There are situations where the test for a condition can be done more efficiently in combination with performing the actual function. In such cases, it is appropriate, for performance reasons, to make the situation regular (that is, not exceptional) and return the condition as a IBoolean result. Consider a function that first tests whether an element exists with a given key, and then accesses it if it exits:

```
if (c.containsElementWithKey (key)) {
     // ...
     myElement = c.elementWithKey (key); // inefficient
     // ...
} else {
     // ...
}
```

This solution is inefficient because the element is located twice, once to determine if it is in the collection and once to access it. Consider the following example:

```
try {
    // ...
    myElement = c.elementWithKey (key); // bad: exception expected
    // ...
} catch (INotContainsKeyException) {
    // ...
}
```

This solution is undesirable because an exception is used to change the flow of control of the program. The correct solution is to obtain a cursor together with the containment test, and then to use the cursor for a fast element access:

```
if (c.locateElementWithKey (key, cursor)) {
    // ...
    myElement = c.elementAt (cursor); // most efficient
    // ...
} else {
    //...
}
```

Levels of Exception Checking

Some preconditions are more difficult to check than others. Consider the following possible preconditions:

1. A cursor for a linked collection implementation still points to an element of a given collection.
2. A collection is not empty.

In the production version of a program, it may be less efficient to check the first precondition than the second.

The Collection Class Library provides three levels of precondition checking. They are selected by the following macro variable definitions (use, for example, compile flag `-DINO_CHECKS`):

`INO_CHECKS` — Check for memory overflow. Other checks may be eliminated to improve performance.

Default — Perform all precondition checks, except the check that a cursor actually points to an element of the collection.

`IALL_CHECKS` — Perform all precondition checks, including the (costly) check that a cursor actually points to an element of the collection. This extra check can only fail for undefined cursors.

List of Exceptions

The Collection Class Library defines the following exceptions:

IChildAlreadyExistsException

Occurs when you try to add a child to a tree using `addAsChild()` at a position that already contains a child.

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.

List of Exceptions

- If a function takes a cursor as an input argument (such as `elementAt()`, `removeAt()`, and `replaceAt()`), the cursor must be *valid*. A cursor is valid if it actually refers to some element contained in the collection. You can use the `isValid()` function to determine if a cursor is valid.

IEmptyException

Occurs when a function tries to access an element of an empty collection. Functions that might cause this exception include `firstElement()` and `removeFirstElement()`.

IFullException

Occurs when a function tries to add an element to a bounded collection that is already full. Functions that might cause this exception include `add()` and `addAsFirst()`.

IIdenticalCollectionException

Occurs when the function `addAllFrom()` is called with the source collection being the same as the target collection.

IInvalidReplacementException

Occurs when, during a `replaceAt()` function, the replacing element has different positioning properties (see “Replacing Elements” on page 95) than the positioning properties of the element to be replaced.

IKeyAlreadyExistsException

Occurs when a function attempts to add an element to a map or sorted map that already has a different element with the same key. Functions that might cause this exception include `add` and `addAllFrom()`.

INotBoundedException

Occurs when the function `maxNumberOfElements()` is applied to a collection that is not bounded.

INotContainsKeyException

Occurs when the function `elementWithKey()` is applied to a collection that does not contain an element with the specified key.

IOutOfMemory

Occurs when a function cannot obtain the space that it requires. This exception is not the result of a precondition violation. Functions that add an element to a collection, including `add()` and `addAsFirst()`, can cause this exception.

IPositionInvalidException

Occurs when a function specifies a position that is not valid in a collection. The functions that might cause this exception include `elementAtPosition()`, `removeAtPosition()`, and `setToPosition()`.

IRootAlreadyExistsException

Occurs when the function `addAsRoot()` is called for a tree that already has a root.

The Hierarchy of Exceptions

In the Collection Class Library, all exceptions are derived from the `IException` class described in Chapter 18, "Exception and Trace Classes" on page 213. It provides common functions to access information about an exception that has occurred.

The direct subclasses of `IException` used in the Collection Class Library are `IPreconditionViolation` and `IResourceExhausted`. The following figure shows the hierarchy of exceptions:

Exception Hierarchy

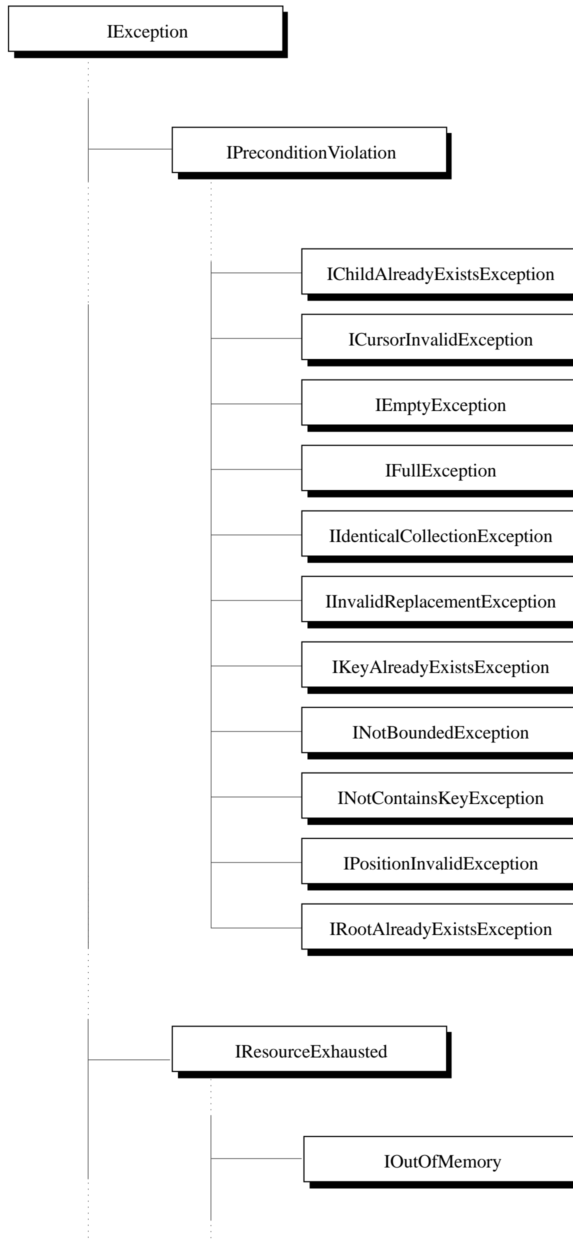


Figure 23. Hierarchy of Exceptions

Chapter 13. Collection Class Library Tutorials

This chapter provides a set of tutorial lessons that you can use to learn common Collection Class Library features. Each lesson builds on the lessons you learned and the library features demonstrated in prior lessons. A section at the end of the chapter describes other tutorials provided with the Collection Class Library that can help you with specific Collection Class Library techniques. Use this chapter if you are beginning to use the library and are unclear on some of the concepts described in earlier chapters of this section.

The lessons in this chapter demonstrate the following capabilities of the Collection Class Library:

- Defining a simple collection
- Adding, removing, and iterating over elements
- Changing the element type
- Changing the collection
- Changing the default implementation

Each lesson has the following format:

- *What the lesson covers*: What you will learn from the lesson.
- *Requirements*: What capabilities must be built into or added to the program.
- *Setup*: What files you will need from previous lessons.
- *Implementation*: Step-by-step instructions for implementing the program requirements. The implementation section includes the required code as well as detailed descriptions of each aspect of the implementation.
- *Source files*: Source file listings showing the contents of, or the order of declaration of functions within, individual source files. Where a source file is not changed from one lesson to the next, it is not listed a second time.
- *Running the program*: A description of what happens when you run the program, observations on the program's behavior, and guidance on optional ways of enhancing or changing the program.
- *What you have learned*: A summary of the Collection Class features that were covered by the lesson.

There are five lessons in this chapter. The following provides an overview of the characteristics of the program used in each lesson, and the Collection Class features the lesson demonstrates:

Lesson 1 A program that builds a collection of integer elements, and adds three elements to the collection. Nothing is done with the collection after these elements are added, and the program produces no output. This lesson demonstrates how to define the element and collection types with **typedef**s, how to instantiate a collection, how to add elements to a collection, and how to determine what Collection Class header file to include.

© Copyright IBM Corp. 1995, 1996

Lesson 2 An enhancement to Lesson 1 that implements a menu so that you can add, list, or remove items, show stock information, or exit the program. Not all these functions are fully implemented at this point. The lesson demonstrates how to iterate over a collection and how to remove elements from a collection.

Lesson 3 An enhancement to Lesson 2 that changes the element type from a built-in type to a class type. The lesson demonstrates how to construct a collection whose elements are of class type, how to determine what element type functions are required, and how to define those functions.

Lesson 4 In this lesson, you change Lesson 3 to use a different collection. The lesson demonstrates how to choose the correct collection for a given application, how to implement various element and key functions, how to use a cursor to iterate through elements with a given key, and how to count the number of elements with a given key.

Lesson 5 In this lesson, you change the implementation *variant* of the collection. This does not change the program's external behavior but in real applications changing an implementation variant can affect performance.

Preparing for the Lessons

To set up the lessons, create five directories beneath the same parent directory, and name them `lesson1` through `lesson5`. You will use these directories to store the files you create for each lesson.

Compiling the Lessons

To compile the lessons, use the following at the OS/2 command line:

```
iccas -Iinclude_path1 ... -Iinclude_pathn /B"crtpgm pgm(mylib/main)
module(mylib/main)" main.c
```

where *include_pathx* is the path for included `.h` and `.hpp` files.

Note: The compiler creates a directory `tempinc` in the directory that is the current directory when the program is compiled. This directory is used by the Collection Class Library to place template files used to instantiate collections in your program. You can delete the files in `tempinc` and the directory itself after compilation.

If the compiler produces errors during compilation, check to make sure that you have specified the required library and that you have typed the source code in correctly. Some common errors are misplacing semicolons and failing to close braces or brackets.

Lesson 1: Defining a Simple Collection of Integers

In this lesson, you write a program that builds a very simple collection of integer elements and adds some elements to the collection. This lesson covers the following Collection Class topics:

- Using a **typedef** to define the element type
- Using a **typedef** to define the collection type
- Instantiating the collection
- Adding elements to the collection
- Specifying the Collection Class header file to include

Requirements

The collection must consist of elements of an integer type. The integer type is to be known as type `Bicycle`, so that later lessons can change the members of the type. The program adds three integers to the collection. Their values are unimportant. The collection is to be a bag.

Setup

Change to the `lesson1` directory, and use an editor to create and edit two files:

`bike.h` This file will contain declarations and typedefs for the element and collection types.

`main.C` This file will contain the **main()** function.

Implementation

The implementation should use **typedef**s to define the element and collection types, so that if the element or collection type changes later, the changes will be automatically reflected in any code that uses the **typedef**.

Defining the Element Type: Use a **typedef** to define a `Bicycle` as a synonym for an `int`. By using a **typedef**, you make it easier to change the element type later, without having to change anything outside the element's type (or class) definition:

```
// in bike.h
typedef int Bicycle;
```

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 `Bicycle`s. By using a **typedef**, you make it easier to change the collection type later, without having to change other parts of your code:

Lesson 1: Defining a Simple Collection

```
typedef IBag <Bicycle> MyCollectionType;
```

In this **typedef**, `IBag` is the default implementation for a bag, `Bicycle` is a template argument representing the element type, and `MyCollectionType` is the type name given to the type being defined (a bag of `Bicycle` elements).

Instantiating a Collection: Now that you have defined a **typedef** for both the element and the collection types, you can instantiate a collection with a type specifier and a name:

```
MyCollectionType MyCollection;
```

Place this definition at global scope so that all functions, not only the `main()` function defined in the next step, have access to the collection and its members. Functions other than `main()` are defined in subsequent lessons.

Adding Elements: You can use Chapter 16, "Flat Collection Member Functions" in the *IBM VisualAge for C++ for AS/400 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 **typedef**s or instantiations that use Collection Classes, you must include the header file for any collection you are using. The chapter on bags in the *IBM VisualAge for C++ for AS/400 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 17, “Bag” in the *IBM VisualAge for C++ for AS/400 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 **typedef**s, how to instantiate a collection and elements, and how to add elements to that collection.

Lesson 2: Adding, Listing, and Removing Elements

The first lesson showed you how to create a simple collection and add three elements. This lesson moves the code for adding elements to a separate function, and implements functions for listing and removing elements as well. These functions are

called from a main program that dispatches the appropriate function based on the user's choice of a menu option.

This lesson covers the following Collection Class topics:

- Iterating over a collection using 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 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` directory to the `lesson2` directory, and then change your current directory to the `lesson2` directory. You will also create two other files. The three files for this tutorial are:

`bike.h` Contains the element and collection typedefs.

`lesson.C` Contains functions for adding, removing, listing, and showing stock information on items.

`main.C` Contains the main menu for the program.

Implementation

You need to replace the body of the `main()` function with the menu handling and function dispatching code. You will make use of I/O Stream input and output to implement the functions that add, list, or remove items. One advantage of using the I/O Stream classes instead of functions like `printf()` and `scanf()` is that, when the element type is changed, you can define input and output operators for the type, and the I/O Stream input and output functions will continue to work without change.

Including the iostream.h Header File: You should include `iostream.h` at the start of `lesson.C` so that you can use the `cin`, `cout`, and `cerr` streams that are predefined by the `iostream` class. You should also include the header file `bike.h` so that you can access the `Bicycle` class and associated functions.

Lesson 2: Adding, Listing, and Removing Elements

```
#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 *IBM VisualAge for C++ for AS/400 IBM Open Class Library Reference*.) Any function invoked by `allElementsDo()` must have a return type of `IBoolean`, and must have two arguments: a **const** reference to the argument and a pointer to void. The pointer to void is used to pass additional arguments to the applied function, if required by the function. For the printing function in this lesson you do not need to pass additional arguments, because the function does not use them. In such cases you pass a `void*` second argument:

```
// in lesson.C
IBoolean printItem (Bicycle const& bike, void* /* Not used */) {
   cout << bike << endl;
   return true;
}
```

The `printItem()` function should always return `true` because it should display the value of each element of the collection. If you wanted certain values of elements to cause

printing to halt, you would have the function return `false` for any such element. A return value of `false` causes the `allElementsDo()` function to stop iterating over the collection.

Just as there was no need to define an input operator for `Bicycle`, there is no need to define an output operator either, as long as `Bicycle` represents an `int`.

Now define the function `listItems()` to call the `printItem()` function for each element of the collection. Use the `allElementsDo()` function for the collection, and use the `printItem()` function as argument. `allElementsDo()` then calls the function for every element of the collection.

```
// in lesson.C
void listItems() {
   MyCollection.allElementsDo( printItem );
}
```

Removing Items: To remove an element from a collection, you need to use the `remove()` member function. This function is described in Chapter 16, “Flat Collection Member Functions” in the *IBM VisualAge for C++ for AS/400 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;
   while (cin.fail()) {
      cin.clear();
      cin.ignore(1000,'\n');
      cerr << "Input error, please re-enter: ";
      cin >> tbike;
   }
   if (!MyCollection.remove(tbike))
      cerr << "Item not found!\n";
}
```

Showing Stock Information: For now, you can define this function to display an error message without changing the collection:

```
// in lesson.C
void showStock() {
   cerr << "Function not implemented yet!\n";
}
```

Main Menu: Finally, change the code in `main()` to display the menu items, accept input, and take appropriate action. Because this code will remain relatively unchanged for subsequent lessons, place it in a separate file, `main.C`, and include `lesson.C` before the code of `main()`. A possible version of `main.C` is shown below.

Source Files for Lesson 2

You should have two source files defined at this point. Their names and sample contents are:

main.C

```
#include <iostream.h>
#include <stdlib.h> // for use of exit() function
void addItem(), listItems(), showStock(), removeItem();

void main() {
   enum Choices { Add, List, Stock, Remove, Exit };
   int menuChoice=0;
   char* menu[5] = {"Add an item",
                    "List items",
                    "Show stock information",
                    "Remove an item",
                    "Exit" };
    while (menuChoice!=5) {
      cout << "\n\n\nSimple Stock Management System\n\n";
      for (int i=0;i<5;i++)
         cout << i+1 << ". " << menu[i] << '\n';
      cout << "\nEnter a selection (1-5): ";
      cin >> menuChoice;
      while (cin.fail()) {
         // get input again if nonnumeric was entered
         cin.clear();
         cin.ignore(1000,'\n');
         cerr << "Enter a selection between 1 and 5!\n";
         cin >> menuChoice;
         }
      switch (menuChoice) {
         case 1: addItem();    break;
         case 2: listItems();  break;
         case 3: showStock();  break;
         case 4: removeItem(); break;
         case 5: exit(0);
         default: cerr << "Enter a selection between 1 and 5!\n";
      }
   }
}
```

lesson.C

```
// lesson.C
#include <iostream.h>
#include <ibag.h>
#include "bike.h"

void addItem() {
   Bicycle tbike;
   cout << "Enter item: ";
   cin >> tbike;
   while (cin.fail()) {
      cin.clear();
      cin.ignore(1000,'\n');
```

```
      cerr << "Input error, please re-enter: ";
      cin >> tbike;
   }
   MyCollection.add(tbike);
}

IBoolean printItem (Bicycle const& bike, void* /* Not used */) {
   cout << bike << endl;
   return true;
}

void listItems() {
   MyCollection.allElementsDo( printItem );
}

void removeItem() {
   Bicycle tbike;
   cout << "Enter item to remove: ";
   cin >> tbike;
   while (cin.fail()) {
      cin.clear();
      cin.ignore(1000,'\n');
      cerr << "Input error, please re-enter: ";
      cin >> tbike;
   }
   if (!MyCollection.remove(tbike))
      cerr << "Item not found!\n";
}

void showStock() {
   cerr << "Function not implemented yet!\n";
}
```

Running the Program

Compile `main.C` and `lesson.C`, link them, and run the program. You can enter elements into the collection, list the elements, remove them, or exit from the program. If you select the option to display stock information, an error message is displayed and no action is taken.

Elements appear to be ordered: If you enter more than one integer into the collection, and then list the collection's elements, you may find that the collection has been sorted from the smallest to the largest element. Do not rely on this ordering relation, because a `Bag` is an unordered, unsorted collection, and changes to your code or to the Collection Class Library could change the order in which elements are accessed.

Multiple equal elements are supported: If you add the number 7 to the collection three times and list the items, the number 7 appears three times. If you then remove the number 7 once, the number 7 still appears twice. A bag supports multiple equal elements.

What You Have Learned

This lesson showed you how to use the `allElementsDo()` function to iterate over elements of a collection, and how to provide a function to `allElementsDo()` that is called for each iterated element. The lesson also demonstrated how to use the `remove()` function to remove elements from a collection.

Lesson 3: Changing the Element Type

Now that you have a working program that allows you to add, list, or remove elements from a collection, you are ready to change the element type to something more complex than an integer.

This lesson covers the following Collection Class topics:

- Defining an element type as a class
- Determining what element type functions are required
- Defining those element type functions

Requirements

The element type must be changed from the built-in integer type to a class type with the following data members:

- A string representing the manufacturer or make of the bicycle
- A string representing the model of the bicycle
- An integer representing the type of bicycle: racing, touring, or mountain bike
- An integer representing the price of the bicycle

Setup

Copy the files `bike.h`, `lesson.C`, and `main.C` from the `lesson2` directory to the `lesson3` directory, and then change your current directory to the `lesson3` directory. Use an editor to modify these files, and to create a new file `bike.C`, which will contain function definitions for functions declared in `bike.h`.

Implementation

First move the **typedef** definition for the collection and the **#include** statement for `ibag.h` from `bike.h` to `lesson.C`, where they are actually made use of.

You can use the `IString` class to handle the strings for make and model. This class includes operators for element equality, ordering relation, and addition (concatenation), all of which will be used in this or later lessons.

Defining the Element Type: In keeping with good object-oriented programming practice, you should separate the member function definitions from the class definition, by placing the class definition in `bike.h` and the definitions of member functions in `bike.C`. You should compile each `.C` file separately, and link them together.

Class Data Members: The following code defines the data members of `Bicycle`. You should replace the **typedef** for the element with the declaration for class `Bicycle`. Two header files are also included because they are required by members of the class. Place the following code in `bike.h`.

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;
// ... 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 *IBM VisualAge for C++ for AS/400 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).

Default Constructor: The default constructor should initialize all data members to blank strings or zero integers:

```
// in bike.h, within class declaration
Bicycle() : Make(""), Model(""), Type(0), Price(0) {}
```

Assignment Operator and Destructor: There is no need to define these explicitly. The compiler generates a default assignment operator and destructor that are suitable for the program.

Copy Constructor: This function is used by the Collection Classes and by the input operator. Declare and define it as follows:

```
// in bike.h:
Bicycle(IString mk, IString md, int tp, int pr) :
   Make(mk), Model(md), Type(tp), Price(pr) {}
```

Equality Test: The equality test (`operator==`) should return `true` if two bicycles have the same make and model, and `false` if not:

```
// in bike.h:
IBoolean operator== (Bicycle const& b) const;

// in bike.C:
IBoolean Bicycle::operator== (Bicycle const& b) const
   { return ((Model==b.Model) && (Make==b.Make)); }
```

Ordering Relation: The ordering relation (`operator<`) should indicate whether the first bicycle would appear before or after the second bicycle in an alphabetically sorted list:

```
// in bike.h:
IBoolean operator< (Bicycle const& b) const;

// in bike.C:
IBoolean Bicycle::operator< (Bicycle const& b) const
   { return ((Make<b.Make) || (Make==b.Make && Model<b.Model)); }
```

You can use the `<` and `==` operators for `IString` objects because they are defined for the `IString` class to indicate alphanumeric sorting order.

Input Operator: This operator is required by the `addItem()` and `removeItem()` functions defined previously. Both this and the output operator are declared *outside* the class definition, at the bottom of `bike.h`, and they are defined in `bike.C`. The input operator stores the alphanumeric data members of `Bicycle` in `char` arrays to avoid the overhead of constructing temporary `IString` objects.

```
// in bike.h:
istream& operator>> (istream& is, Bicycle& bike);

// in bike.C:
istream& operator>> (istream& is, Bicycle& bike) {
   char make[40], model[40];
   char typeChoice;
   float price;
   int type=-1;
   cin.ignore(1,'\n'); // ignore linefeed from previous input
   cout << "\nManufacturer: ";
   cin.getline(make, 40, '\n');
   cout << "Model: ";
   cin.getline(model, 40, '\n');
   while (type == -1) {
      cout << "Racing, Touring, or Mountain Bike (R/T/M): ";
      while (cin.fail()) {
         cin.clear();
         cin.ignore(1000,'\n');
         cerr << "Racing, Touring, or Mountain Bike (R/T/M): ";
         cin >> typeChoice;
      }
      switch (typeChoice) {
         case 'r':
         case 'R': { type=Racing; break; }
         case 't':
         case 'T': { type=Touring; break; }
         case 'm':
         case 'M': { type=MountainBike; break; }
         default:  { cerr << "Incorrect type, please re-enter\n"; }
      }
   }
   cout << "Price ($$.$$): ";
   cin >> price;
   while (cin.fail()) {
      cin.clear();
      cin.ignore(1000,'\n');
      cerr << "Enter a numeric value: ";
      cin >> price;
      }
   price*=100;
   bike=Bicycle(make,model,type,price);
   return is;
}
```

Output Operator: The output operator is required by the `listItems()` function, and may later be required by other functions. It should display the make, model, type, and price of a bicycle:

```
// in bike.h:
ostream& operator<< (ostream& os, Bicycle bike);

// in bike.C:
ostream& operator<< (ostream& os, Bicycle bike) {
   return os << bike.Make
             << "\t" << bike.Model
             << "\t" << btype[bike.Type]
             << "\t" << float(bike.Price)/100;
}
```

Source Files for Lesson 3

The program should now be placed in the following files. Some function bodies have been replaced with ellipses for brevity. `main.C` remains unchanged and is not shown.

lesson.C

```
// lesson.C
#include <iostream.h>
#include <ibag.h>
#include "bike.h"
typedef IBag<Bicycle> MyCollectionType;
MyCollectionType MyCollection;

void addItem() { /* ... */ }
IBoolean printItem (Bicycle const& bike, void* /* Not used */)
   { /* ... */ }
void listItems() { /* ... */ }
void removeItem() { /* ... */ }
void showStock() { /* ... */ }
```

bike.h

```
#include <istring.hpp>  // access to IString class
#include <iostream.h> // access to iostream class

class Bicycle {
   public:
   IString Make;
   IString Model;
   int Type;
   int Price;
   Bicycle() : Make(""), Model(""), Type(0), Price(0) {}
   Bicycle(IString mk, IString md, int tp, int pr) :
   Make(mk), Model(md), Type(tp), Price(pr) {}
   IBoolean operator== (Bicycle const& b) const;
   IBoolean operator< (Bicycle const& b) const;
};
istream& operator>> (istream& is, Bicycle& bike);
ostream& operator<< (ostream& os, Bicycle bike);
```

Lesson 3: Changing the Element Type

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` directory to the `lesson4` directory, and then change your current directory to the `lesson4` directory. Use an editor to modify the files as described below.

Implementation

The collection must have the following characteristics:

Key access, so that an element can be accessed using only its make and model information (for the listing and removing functions)

No element order, because order is not specified as a requirement

Multiple elements with the same key, so that several bicycles of the same make and model can be present in the collection

Element equality, so that elements with the same make and model can have different price and type information

You can use Figure 7 on page 79 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):

Lesson 4: Changing the Collection

```
// in bike.h:
class Bicycle {
      IString MMKey; // add a private data member for the key
   public:
      // public data members and member functions
      Bicycle(IString mk, IString md, int tp, int pr);
      // ...
};

// in bike.C:
Bicycle::Bicycle(IString mk, IString md, int tp, int pr) :
        Make(mk), Model(md), Type(tp), Price(pr),
        MMKey(mk+md) {}
```

Defining Key Access: The key access function must be defined *outside* of the element class. It has one argument, whose type is the element type. The key access function must call a member function that returns the key, in this case a function named `getKey()`. (The actual name does not matter.) The member function accesses the private data member `MMKey`.

```
// in bike.h:
class Bicycle {
      IString MMKey;
   public: // ... data members and member functions
      IString const& getKey() const;
   };

inline IString const& key (Bicycle const& bike)
   { return bike.getKey(); }

// in bike.C:
IString const& Bicycle::getKey() const { return MMKey; }
```

The key access function must be declared with the name `key()`, with a **const** reference to the key as its return value, and a **const** reference to the element as its argument.

Equality Test: Equality for elements should be defined such that the key (that is, the make and model), the type, and the price are the same for two bicycles. The `operator==` function in `bike.C` can be redefined as follows:

```
IBoolean Bicycle::operator== (Bicycle const&b) const {
    return (MMKey==b.MMKey && Type==b.Type && Price==b.Price);
    }
```

Key Hash Function: The hash function provides a shortcut for Collection Class search functions to find matches to a key. The search functions first call the hash function on a key for which they need to locate an element. They use the hash value returned to look for matches to that hash in a hash table. They then use the full key to determine which of the hash function's matches have the correct key. The hash key-type function is not a member function of the element's class. It is called by the searching function, with a key argument (the key on which to derive the hash) and an unsigned long (the maximum hash value). The return value is the hash, and it cannot exceed the maximum hash value. The hash function should be defined in `lesson.C` and must have the following return type and parameters:

```
unsigned long hash ( IString const& keyName,    unsigned long hashInput );
```

You can define the hash using the hashing function provided in istdops.h for char* values:

```
unsigned long hash (IString const &aKey, unsigned long hashInput) {
     return hash( (const char*)aKey, hashInput);
     }
```

Using Cursors to Remove Items: A Collection Class cursor (not related to the cursor used to move about a cursor screen) is a reference to an element in a collection. For an overview of cursors, see "Cursors" on page 95.

The removeItem() function must be redefined so that it requests the make and model of bicycle to remove, lists all matching bicycles, and lets the user choose which match to remove. Once matching bicycles have been displayed, a cursor can be used to locate the bicycle the user wishes to delete. The cursor is defined as follows, immediately after the collection MyCollection is declared, in lesson.C:

```
MyCollectionType::Cursor thisOne (MyCollection);
```

After the user enters a make and model to search for, the removeItem() function should iterate through all elements that match the key, by using locateElementWithKey() to find the first matching element, and locateNextElementWithKey() to find all subsequent matching elements. Both these functions require a cursor as their second argument, and the cursor points to the located element when the functions return. The first part of removeItem() can be redefined as follows:

```
void removeItem() {
   Bicycle tbike;
   int choice, cursct=1;
   cout << "\nRemove an item";
   cin >> tbike;
   if (MyCollection.numberOfElementsWithKey(tbike.getKey()) > 0) {
      MyCollection.locateElementWithKey(tbike.getKey(), thisOne);
      cout << cursct << ". " << MyCollection.elementAt(thisOne) << endl;
      for ( cursct=2;
            MyCollection.locateNextElementWithKey(
                        tbike.getKey(), thisOne);
            cursct++)
            {  cout << cursct << ". "
                    << MyCollection.elementAt(thisOne) << endl; }
   //... Remainder to be defined later
   }
```

In the above fragment, the user is asked for a bicycle make and model to remove. If any elements match the make and model (this is determined by testing the numberOfElementsWithKey() function for a nonzero return), all such elements are located by key. The locateElementWithKey() function sets its cursor to point to the first matching element, and the locateNextElementWithKey() function advances the cursor from the current match to the next match in the collection. The elements are accessed for output using the elementAt() function, which returns a reference to the element pointed to by the cursor argument.

Once the matching elements have been displayed with a number beside each one, the program should ask the user to enter a number matching the number of the element to

remove. The matching elements can then be iterated over again until the number of elements iterated over matches the user's selection, and the element pointed to by the cursor is then deleted. The following code excerpt is the second part of the `removeItem()` function:

```
// Insert this at "...Remainder to be defined later" in removeItem().
    cout << "\nEnter item to remove, or 0 to return: ";
    cin >> choice;
    if (choice<=0 || choice > cursct) return;
    MyCollection.locateElementWithKey(tbike.getKey(),thisOne);
                 // locate the first matching element again
    for ( cursct=2;
          cursct<=choice &&
          MyCollection.locateNextElementWithKey    // check for valid
                       (tbike.getKey(), thisOne);  // next match
          cursct++)
          ; // null loop - header contains the code to be executed
    MyCollection.removeAt(thisOne);
    }
 else
    cerr << "No bicycles of this make and model were found.\n";

// The closing brace below was already part of removeItem().
// Do not duplicate it.
 }
```

Note: The `locateNextElementWithKey()` function invalidates the cursor if it cannot find a next element with the key provided. An invalidated cursor does not point to any element of the collection. Some flat collection member functions that use cursors require that the cursor be valid (`locateNextElementWithKey()` is one such function). Before you use a cursor with such a function, you need to validate the cursor by using a function that takes a cursor as argument but does not require a valid cursor on entry. `locateElementWithKey()` is one such function.

In both excerpts of `removeItem()` above, the elements with matching keys are iterated over by code in the header of the loop. In the second case, the loop has no body. You can use this coding style because all the `locate...` functions have a return type of `IBoolean`, which can be used in condition tests such as those in loop control expressions.

Showing Stock Information: `showStock()` must be rewritten so that, for a given make and model, it displays the number of matching elements in the collection. The `numberOfElementsWithKey()` function can be used:

```
void showStock() {
   Bicycle tbike;
   int count;
   cout << "Stock information for a model";
   cin >> tbike;
   count=MyCollection.numberOfElementsWithKey(tbike.getKey());
   if (count!=1)
      cout << "Currently there are " << count << " bicycles ";
   else
      cout << "Currently there is 1 bicycle ";
   cout << "of this make and model in stock." << endl;
   }
```

Lesson 4: Changing the Collection

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;
    }
```

Lesson 4: Changing the Collection

```
// Comment out the ordering relation operator
// IBoolean Bicycle::operator< (Bicycle const& b) const
//    { return ((Make<b.Make) || (Make==b.Make && Model<b.Model)); }
IBoolean Bicycle::operator== (Bicycle const&b) const {
  return (MMKey==b.MMKey && Type==b.Type && Price==b.Price);
  }
IString const& Bicycle::getKey() const { return MMKey; }

istream& operator>> (istream& is, Bicycle& bike) {
    char make[40], model[40];
    char typeChoice;
    float price=0;
    int type=-1;
    cin.ignore(1,'\n'); // ignore linefeed from previous input
    cout << "\nManufacturer: ";
    cin.getline(make, 40, '\n');
    cout << "Model: ";
    cin.getline(model, 40, '\n');
    bike=Bicycle(make,model,type,price);
    return is;
 }

ostream& operator<< (ostream& os, Bicycle bike) {/* ... */} // unchanged
```

lesson.C

```
// lesson.C
#include <iostream.h>
#include <irel.h> // was ibag.h
#include "bike.h"
enum bikeTypes { Racing, Touring, MountainBike };
typedef IRelation<Bicycle,IString> MyCollectionType;

MyCollectionType MyCollection;
MyCollectionType::Cursor thisOne (MyCollection);

IBoolean printItem (Bicycle const& bike, void* /* Not used */)
   { /* ... */ }

void addItem() {
   Bicycle tbike;
   char typeChoice;
   float price;
   int type=-1;
   cout << "Enter item: ";
   cin >> tbike;
   while (type == -1) {
      cout << "Racing, Touring, or Mountain Bike (R/T/M):";
      cin >> typeChoice;
      while (cin.fail()) {
         cin.clear();
         cin.ignore(1000,'\n');
         cerr << "Racing, Touring, or Mountain Bike (R/T/M): ";
         cin >> typeChoice;
         }
      switch (typeChoice) {
         case 'r':
         case 'R': { type=Racing; break; }
         case 't':
         case 'T': { type=Touring; break; }
         case 'm':
         case 'M': { type=MountainBike; break; }
         default:  { cerr << "Incorrect type, please re-enter\n"; }
      }
   }
   cout << "Price ($$.$$): ";
   cin >> price;
```

```
    price*=100;
    tbike.Type=type;
    tbike.Price=price;
    MyCollection.add(tbike);
 }

 void listItems() {/* ... */ }
 void removeItem() {
    Bicycle tbike;
    int choice, cursct=1;
    cout << "\nRemove an item";
    cin >> tbike;
    if (MyCollection.numberOfElementsWithKey(tbike.getKey()) > 0) {
       MyCollection.locateElementWithKey(tbike.getKey(), thisOne);
       cout << cursct << ". " << MyCollection.elementAt(thisOne) << '\n';
       for ( cursct=2;
             MyCollection.locateNextElementWithKey(
                                  tbike.getKey(), thisOne);
             cursct++)
             {  cout << cursct << ". "
                      << MyCollection.elementAt(thisOne) << '\n'; }
       cout << "\nEnter item to remove, or 0 to return: ";
       cin >> choice;
       if (choice<=0 || choice > cursct) return;
       MyCollection.locateElementWithKey(tbike.getKey(),thisOne);
                    // locate the first matching element again
       for ( cursct=2;
             cursct<=choice &&
             MyCollection.locateNextElementWithKey    // check for valid
                          (tbike.getKey(), thisOne);  // next match
             cursct++)
             ; // null loop - header contains the code to be executed
       MyCollection.removeAt(thisOne);
       }
    else
       cerr << "No bicycles of this make and model were found.\n";
    }

 void showStock() {
    Bicycle tbike;
    int count;
    cout << "Stock information for a model";
    cin >> tbike;
    count=MyCollection.numberOfElementsWithKey(tbike.getKey());
    if (count!=1)
       cout << "Currently there are " << count << " bicycles ";
    else
       cout << "Currently there is 1 bicycle ";
    cout << " of this make and model in stock." << endl;
    }

unsigned long hash (IString const &aKey, unsigned long hashInput) {
     return hash( (const char*)aKey, hashInput);
     }
```

Running the Program

You can enter multiple bicycles of the same make and model, with different price or type information, and all such models will appear when you select the “List items” option. When you ask for stock information, the program displays the number of elements in the collection that match the make and model information you specify. When you remove an item, the program asks you for the make and model, displays a list of matching items, and lets you specify which item to remove. The program removes that item.

What You Have Learned

The Collection Class Library offers a wide range of collections with different characteristics. In this lesson, you learned how to select an appropriate collection based on the characteristics of the data being placed in the collection and on the intended uses of the data. Many Collection Classes are accessed or sorted using a key, and you learned how to define key access, equality, and hash functions, and how to iterate through a key collection using a key cursor.

Lesson 5: Changing the Implementation Variant

You should pursue changing the default implementation to an implementation variant only after the program is functionally complete and has been fully debugged. The purpose of changing to a nondefault implementation variant is to improve performance. This lesson shows you how to change the code defined in "Lesson 3: Changing the Element Type" on page 159 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 an AVL tree implementation. Other implementation variants are:

- Bag as B* Tree
- Bag as List
- 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.

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.

Lesson 5: Changing the Implementation Variant

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 *IBM VisualAge for C++ for AS/400 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 17, “Bag” in the *IBM VisualAge for C++ for AS/400 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 th e required additional function is a hash function, which is already described in “Lesson 4: Changing the Collection” on page 165.

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 159. In a complex application, a change in performance might occur, but in all cases the behavior of a correctly coded

program should be identical for different implementation variants of the same collection class.

What You Have Learned

Once a C++ program using the Collection Classes is functionally complete and error-free, you can focus on performance. The key to good performance of Collection Classes programs is to select the appropriate implementation variant of a given collection. Although this lesson did not explain which implementation variant to choose (since this is largely dependent on the class type being used in the collection and on other factors beyond the scope of the lessons), it showed you how to change the implementation variant once the appropriate variant has been selected. See “Features of Provided Implementation Variants” on page 130 for guidance on what implementation variants to select for a given application.

Errors When Compiling or Running the Lessons

If you code the programs in this chapter exactly as shown, they should compile successfully, and should run without any errors except those related to incorrect user input. Check your code for typographical mistakes or incorrectly placed code if you get compiler errors.

If you implement element, key, input, or output functions in different ways than those indicated, and your program does not compile successfully, or compiles but ends with an exception message when run, you can use Chapter 14, “Solving Problems in the Collection Class Library” on page 179 to determine the cause. You can also use Chapter 14 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 the VisualAge for C++ for AS/400 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 91)

Tutorial 2 Adding, removing and replacing elements in a collection ("Adding, Removing, and Replacing Elements" on page 92)

Tutorial 3 Use of a cursor, locating and accessing elements, and the use of applicators ("Cursors" on page 95, "Using Cursors for Locating and Accessing Elements" on page 96, "Iterating over Collections" on page 98)

Tutorial 4 Use of exceptions (Chapter 12, "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 127)

Tutorial 6 Using abstract base classes to write polymorphic functions (Chapter 11, "Polymorphism and the Collections" on page 139)

Source Files for the Tutorials

Every directory contains the following files:

`tutor?.rea`	Read this to understand the purpose of the tutorial.
`tutor?.txt`	Instructions to follow.
`tutor?.mak`	Prepared makefile to compile the example.
`solution`	Directory containing a possible solution.

You will find prepared `.c` and `.h` files, where certain parts are missing. The objective of the tutorials is to apply the information you have learned about the Collection Class Library by adding the missing parts for each file. Complete the prepared files following the instructions in `instruct.txt`. You can compare your solutions to the solutions directory.

Other Tutorials

Chapter 14. 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	179
Element Functions and Key-Type Functions	Error messages indicating a problem in *istdops.h*	180
Key Access Function - How to Return the Key (1)	Error messages indicating a problem in *istdops.h*: a local variable or compiler temporary is being used in a return expression	181
Key Access Function - How to Return the Key (2)	Unexpected results when adding an element to a unique key collection	182
Exception Tracing	Unexpected exception tracing output on standard error	183
Declaration of Template Arguments and Element Functions (1)	Compiler messages (when templates are being processed) indicating that an element type or one of its required element functions is not declared	183
Declaration of Template Arguments and Element Functions (2)	Compilation errors from symbols being defined multiple times	184
Declaration of Template Arguments and Element Functions (3)	Bind errors from symbols being defined multiple times	184
Default Constructor	Compiler error messages indicating a problem with constructors	185

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 95 and "Removing Elements" on page 93.

Element Functions and Key-Type Functions

Effect

When compiled, your program causes a compiler error indicating a problem in *istdops.h*. The following are examples of such errors:

Message if key is missing

```
j:\...\ibmclass\istdops.h(166:1) : (E) CTT3013:
  "key" is undefined.
j:\...\ibmclass\istdops.h(160:1) : informational CTT3207:
  The previous message applies to the definition of template
  "IStdKeyOps<Parcel,ToyString>::key(const Parcel&) const".
```

Message if hash is missing

```
j:\...\ibmclass\istdops.h(152:1) : (E) CTT3070:
  Call does not match any argument list for "::hash".
j:\...\ibmclass\istdops.h(146:1) : informational CTT3207:
  The previous message applies to the definition of template
  "IStdHshOps<ToyString>::hash(const ToyString&,unsigned long) const".
```

Message if == is missing

```
j:\...\ibmclass\istdops.h(81:1) : (E) CTT3054:
  The "==" operator is not allowed between "const ToyString" and
  "const ToyString".
j:\...\ibmclass\istdops.h(80:1) : informational CTT3207:
  The previous message applies to the definition of template
  "equal(const ToyString&,const ToyString&)".
```

Message if < is missing

```
j:\...\ibmclass\istdops.h(105:1) : (E) CTT3054:
  The "<" operator is not allowed between "const ToyString"
  and "const ToString".
j:\...\ibmclass\istdops.h(103:1) : informational CTT3206:
  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 105.

Note that the same problem may be produced if function declarations and definitions are not properly separated between .h files and .C files. This situation is described in detail in "Declaration of Template Arguments and Element Functions (1)" on page 183.

Key Access Function - How to Return the Key (1)

Effect

You get a compiler warning similar to:

Message if key is passed by value

```
j:\...\ibmclass\istdops.h(166:1) : warning CTT3285:
  The address of a local variable or compiler temporary is being used
  in a return expression.
j:\...\ibmclass\istdops.h(160:1) : informational CTT3207:
  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.

Definition of Key-Type Functions

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 105.

Key Access Function - How to Return the Key (2)

Effect

You are adding an element into a unique key collection, such as a key set or a map, and you are sure that the collection does not yet contain an element with the same key. Nevertheless, you get unexpected results: `IKeyAlreadyExistsException`, or the element is not added and the cursor is positioned to a different element.

Reason

This problem has the same cause as the problem described in “Key Access Function - How to Return the Key (1)” on page 181. 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 181.

Definition of Key-Type Functions

Effect

You are using a collection class with a key, and you get an error message during the bind step indicating a problem in *istdops.h*. The following are examples of such errors:

Message if key() function is undefined

```
istdops.h(176): (E) CTT3013: "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 .C file but the .h file with the same name as the .C file does not contain a declaration (prototype) of the global key function.

While compiling the .C file, which uses methods of the collection class, the VisualAge for C++ for AS/400 compiler has created or modified a temporary .C file in the `tempinc` directory. During the bind step, this .C file is compiled to resolve references to template code. The error message you encounter refers to this compilation. The .C file in the `tempinc` directory contains include directives for the collection class template code. It also contains include directives for a .h file of the same name as the .C file that uses

the collection class methods. The template code in *istdops.h* requires that the global `key()` function be known at compilation time. The only file that is included at this time is the .h file with the same name as your .C file. The problem is that the .C file is not included at this time, so a definition or declaration of the global `key()` function in this file is not recognized by the compiler.

Solution

You must declare the global `key()` function in the .h file with the same name as the .C file that uses the collection class methods. The definition of the global `key()` function should be in the .C file. If you are not sure which .h file is meant by the message, look in the .C file found in the *tempinc* directory.

Exception Tracing

Effect

You get unexpected exception tracing output on standard error, even though the related exception causing the output is caught.

Reason

For each exception raised, the trace function `write()` of class `IException::TraceFn` is called and writes information about the raised exception to standard error. This trace function `write()` is called whether the related exception is caught or not.

Solution

To suppress the trace output, provide your own `IException::TraceFn::write()` tracing function by subclassing `IException::TraceFn` and register the subclass with `setTraceFunction()`.

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 .C file that contains the template instantiation with the element type as its argument. The prebind phase is executed only by using the header files. Therefore, your declaration local to a .C file is not recognized and causes these compilation errors.

Solution

Move the corresponding declarations to a separate header file and include the header file from the .C 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 bind 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 .C file or make them inline.

Default Constructor

Effect

You get a compiler error similar to the following:

Message for missing default constructor

```
itbseq.h(25:1) : (E) CTT3222:
"IGTabularSequence<ToyString,IStdOps<ToyString> >::Node" needs a
constructor because class member "ivElement" needs a constructor
initializer.
Names namesOfExtinct(animals.numberOfDifferentKeys());
ANIMALS.C(55:57) : informational CTT3207:
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 105. The element and key-type functions required for each collection are listed for each collection type in sections entitled “Template Arguments and Required Functions.”

Default Constructor

Chapter 15. Compatibility Information

This chapter tells you how the changes to the Collection Class Library can affect existing programs and how you develop, compile, and bind 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 188 identifies situations in which you need to recompile existing applications.

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 9 on page 86, 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.

IIterator, IConstantIterator

The classes `IIterator` 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 `IIterator` 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.

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	ikeyset.h	IKeySet	iks.h
		IKeySetAsAvlTree	iksavl.h
IKeySetOnBSTKeySortedSet	iksbst.h	IKeySetAsBstTree	iksbst.h
IHashKeySet	ihshks.h	IKeySetAsHshTable	ikshsh.h
IKeySetOnSortedLinkedSequence	ikssls.h	IKeySetAsList	ikslst.h
IKeySetOnSortedTabularSequence	ikssts.h	IKeySetAsTable	ikstab.h
IKeySetOnSortedDilutedSequence	ikssds.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 Version 3 Release 7 Modification 0. 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 VisualAge C++ for OS/400 Version 3 Release 6 Modification 0, and are migrating to the Collection Classes that are shipped with VisualAge for C++ for AS/400 Version 3 Release 7 Modification 0 or later, 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 139.

Part 4. The Data Type and Exception Class Library

This part tells you how to use the Data Type and Exception classes. You can use these classes to create and manipulate strings, date and time information, handle exceptions, or define your own classes.

Chapter 16. Data Types and Exceptions 191
Organization of Classes 191
IBase Class 194
IVBase Class 194
String and Buffer Classes 195
DBCS and National Language Support 195

Chapter 17. String Classes 197
Introduction to the String Classes 197
What You Can Do with Strings 198
IStringTest Class 210

Chapter 18. Exception and Trace Classes 213
Introduction to the Exception Classes 213
Catching Exceptions Thrown by Class Library Functions 215
Throwing Your Own Exceptions Using the Exception Classes 216
Macros Used with the Exception Classes 217
Using the ITrace Class 220

Chapter 19. Date and Time Classes 225
IDate Class 225
ITime Class 227
ITimeStamp Class 229

Chapter 20. The IBM Open Class Notification Framework 231
Notifiers and Observers 231
Notification Protocol 233
IBM C++ Notification Class Hierarchy 234

© Copyright IBM Corp. 1995, 1996

Chapter 16. Data Types and Exceptions

The Data Type and Exception 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 "Application Support Class Library."

Organization of Classes

Figure 24 on page 192 shows the organization of the Data Type and Exception classes that are derived from `IBase` and those that are derived from `IException`. Five other classes do not inherit from any classes and are used to support the derived classes. See Table 5 on page 194 for information on the names of these classes and the classes they support. The purposes of the principal classes are described below. Classes are listed alphabetically.

`IBase`
: The base class of most of the other classes in the Data Types, Exception, and User Interface classes of IBM Open Class Library. This class provides an output operator and conversion functions for the library, and **typedef** synonyms used by other library classes to make programming easier. You do not need to create objects of the `IBase` class; it is described for completeness only.

`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.

`IBuffer`
: Objects of the buffer classes contain the actual character contents of objects of the string classes. All manipulation of string characters is done in the buffer object referenced by the string object. `IBuffer` is the buffer class for single-byte character set objects.

`IDate`
: This class provides support for date information. You can construct `IDate` objects in a number of ways, and then use `IDate` methods to determine the day of the week, month or year, compare two dates, test a date for certain characteristics, and obtain the names of days or months that are dependent on the national-language locale setting in effect at run time.

`IDBCSBuffer`
: This class is the buffer class for double-byte character sets. Double-byte character sets are used for handling languages such as Japanese, Chinese, and Korean,

© Copyright IBM Corp. 1995, 1996

Class Organization

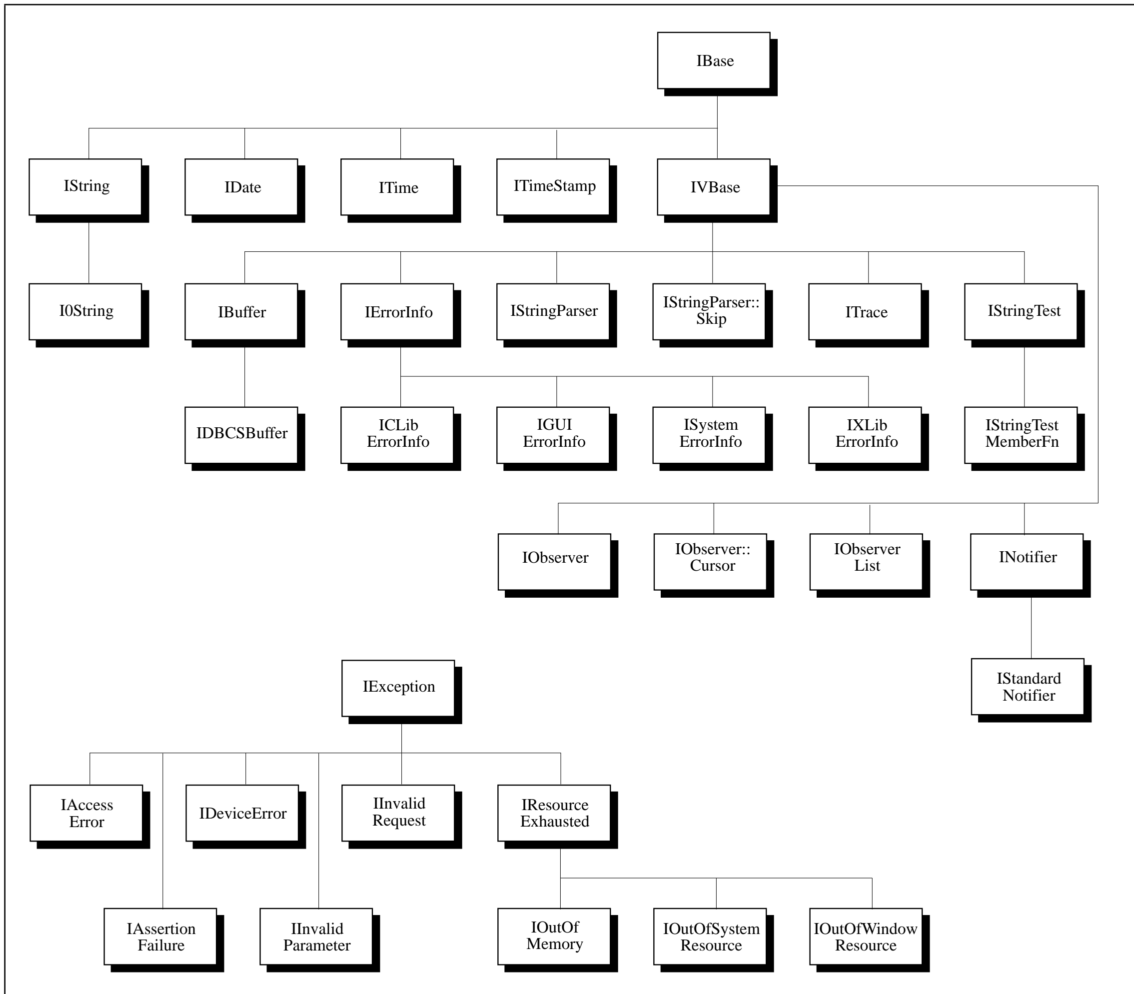


Figure 24. Organization of Data Type and Exception Class Library. Some class names have been split into two lines to fit in their boxes. Note that `IGUIErrorInfo`*,* `IXLibErrorInfo`*, and* `IOutOfSystemResource` *are not supported on VisualAge for C++ for AS/400.*

which contain more symbols than can be represented by the 256 characters of the single-byte character set.

`IException` — The `IException` class is the base class from which all exception objects thrown in the library are derived.

`IObserver` — This class, along with the `IObserverList`, `INotifier`, `IStandardNotifier`, and `IObserver::Cursor` classes, lets you register observers with class objects so that you can be notified when a change to such an object takes place. For further information on using these

classes, see *Building VisualAge C++ Parts for Fun and Profit*.

`IString`
: This class gives you a greater flexibility in handling strings than traditional C-style character arrays. The `IString` class supports both single- and double-byte character sets. With `IString` objects, you can code string-handling operations much more quickly. For example, you can concatenate two strings simply by using the + operator, or compare them using the == operator.

`IStringTest`
: This class is provided so that you can define your own version of the matching function used by `IString` search and compare methods.

`ITime`
: You can use this class to create time-of-day objects and to compare them, add them together, extract specific information from them, or write them to an output stream.

`ITimeStamp`
: 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.

`ITrace`
: Objects of the `ITrace` class provide module tracing. Whenever an exception is thrown by the library, trace records are output with information about the exception.

`IVBase`
: This class is a virtual base class used to derive other classes such as the buffer classes.

`I0String`
: This class is identical to the `IString` class, except in its method of indexing strings. In the `IString` class, the first character of a string is at position 1, whereas the same string when stored in an `I0String` object has its first character at position 0. `I0String` is provided for programmers who are used to the C string-handling approach of treating strings as starting at position 0. `IString` and `I0String` objects are easily interchanged, and they support the same set of methods and operators.

One of the most important classes from a programmer's perspective is the `IString` class. This class can make your programming much more productive if you do any amount of string handling. The `IString` class provides a simpler, safer, and more flexible way of handling strings than traditional C-style character arrays and the functions of the `string.h` library. The `IString` class has associated classes that give you even greater flexibility in how you index strings and in how you test for pattern matches in the searching and replacing functions the class provides.

IVBase Class

Table 5. Support Classes for Data Type and Exception Classes

Class Name	Supports These Classes
IStringEnum	IString I0String IBuffer IDBCSBuffer
IMessageText	IBase
IException::TraceFn	IException
IExceptionLocation	IException

IBase Class

The `IBase` class provides:

- An output operator for the library
- Conversion functions for the library
- 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. If you are using the string classes, DBCS support is automatic and transparent. However, if your code contains DBCS literals, you need to compile with the `Allow DBCS support` compiler option (`/Sn`).

`IBuffer` and `IDBCSBuffer` are purely internal classes used in the implementations of `IString` and `I0String`. They are only used in protected sections of the `IString` class. They are described in this guide because you may want to understand them if you are deriving classes from `IString`.

The support classes, `IStringEnum` and `IStringTest`, provide data types and testing functions that are used in the string and buffer classes.

DBCS and National Language Support

The library provides double-byte character set (DBCS) support and national language support (NLS). You can use one source file for your application code and provide DBCS and NLS support by using separate resource files for the languages you support. The benefits of this organization include the following:

- The application is easy to maintain, because a single version of the application is used. This reduces the cost of maintaining your code.
- The application is easy to upgrade because only the source code is upgraded and then linked to the separate language files for different languages. This reduces the time and cost of upgrading your code because different language versions can be generated at the same time.

Because message strings are defined in message files, they can be translated easily to your local language without changes to the source code.

DBCS

You should note the following when creating a DBCS-enabled application:

- String manipulation is DBCS-enabled. The string classes support mixed strings that contain both SBCS and DBCS characters. Use the string testing functions to determine if a character is single byte or double byte.
- The `IDBCSBuffer` class ensures that the search functions do not match the second or any subsequent bytes of a DBCS character and that the bytes of a DBCS character will not be split.
- If your code contains DBCS literals, you need to compile with the `Allow DBCS support` compiler option (`/Sn`).

DBCS and National Language Support

Chapter 17. String Classes

The string classes define a data type for strings and provide member functions that let you perform a variety of data manipulation and management activities. They provide capabilities far beyond those available with standard C strings and the `string.h` library functions.

The string classes have the following capabilities:

- String buffers are handled automatically.
- Strings can contain both SBCS and DBCS characters.
- Strings can be indexed by character or by word.
- Strings can contain null characters. (There are no restrictions on the contents of a string object.)

Member functions of the string classes allow you to:

- Use strings in input and output
- Access information about strings
- Compare strings
- Test the characteristics of strings
- Search for characters or words within a string
- Manipulate and edit strings
- Convert strings to and from numeric types
- Format strings by adding or removing white space

Introduction to the String Classes

There are two string classes: `IString` and `I0String`. They are identical except for the method each uses to index its characters. The characters of an `IString` object are indexed beginning at 1. `I0String` characters are indexed beginning at 0. See "Indexing of Strings" on page 198 for more information on the indexing of the string classes. The string class you should use depends on which indexing scheme you prefer or find easier to implement.

Objects of `IString` and objects of `I0String` can be freely intermixed in a program. Objects of one class can be assigned objects of the other. Arguments that require an object of one will accept objects of the other. You will only notice a difference between an `IString` and an `I0String` when you are using functions that use or return a character index value.

In this chapter, only the `IString` class is presented. However, for every function of the `IString` class, there is a corresponding and identically named function of the `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 *IBM VisualAge for C++ for AS/400 IBM Open Class Library Reference*.

© Copyright IBM Corp. 1995, 1996

String Buffers

When you create an object of a string class, the actual characters that make up the string are not stored in the string object. Instead, the characters are stored in an object of a buffer class.

The use of a buffer object is transparent to you when using the string classes. A correctly sized buffer is automatically created when you create a string object. The buffer is destroyed when a string object is destroyed. When you manipulate or edit a string, you are actually manipulating and editing the buffer object that contains the characters of the string.

Double-Byte Character Set Support

Objects of the `IString` class and the `I0String` class can contain a mixture of single-byte characters and double-byte characters. All member functions allow for the mixture. The searching functions will not match a single-byte character with the second or subsequent byte of a double-byte character. Functions that return substrings will never separate the bytes of a double-byte character.

Although the double-byte characters are supported, you must be careful not to alter the contents of a string in a way that would corrupt the data. For example, the statement:

```
IString[n]='x';
```

would be an error if the nth byte of the `IString` was part of a double-byte character.

Indexing of Strings

Objects of the string classes are arrays of characters. There are two types of indexes used with the arrays. The first is a character index: each character is numbered from left to right starting at the number 1 in the `IString` class and the number 0 in the `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 `IString`s 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 *IBM VisualAge for C++ for AS/400 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 `IString`s using constructors, and you can copy `IString`s using copy constructors, assignment operators, and substring functions.

IString Constructors

You can use `IString` constructors that construct null strings, that accept a numeric argument and convert it into a string of numeric characters, or that translate one or more characters into an `IString`. You can also create a single string out of up to three separate buffers, whose contents are concatenated into the created `IString` object. The following example shows some of the above ways of creating `IString` objects:

```
#include <istring.hpp>
#include <iostream.h>
void main() {
   IString Number1(123);    // --> Number1   ="123"
   IString Number2(123.12); // --> Number2   ="123.12"
   IString Character('a');  // --> Character ="a"
   IString String1("a");    // --> String1   ="a"
   IString String2("and");  // --> String2   ="and"
   IString String3("a\0d"); // --> String3   ="a"
   }
```

Note that the last string (`String3`) is initialized with only the first byte of quoted text. The null character in the **char*** constructor argument is interpreted by the compiler as a terminating null. However, the `IString` class does support null bytes within strings. To construct `String3` as the example intended, you could write:

```
//...
IString String3("and");
String3[2]='\0';
```

If this string is later copied to another string, the null character and following characters are also copied:

```
IString String4=String3;
String4[2]='N';          // --> String4   ="aNd"
```

Copying IStrings

The IString assignment operator and copy constructor both copy one string to another string. One of the strings can be an array of characters, or both may be IString objects. The IString assignment operator and copy constructor offer the following advantages over the strcpy and strdup functions provided by the C string.h library:

- When an IString object is copied, a new copy of the string is not made. Instead, the two strings point to the same buffer location. The object is only copied if one of the strings is changed. This means that, for strings that are copied but where neither the source string nor the copy is subsequently changed, performance is improved by the amount of time it would have taken to make the new copy.
- The notation is simple and intuitive. To copy String1 into String2, you simply code String2=String1. With strings defined using the traditional **char*** method, such an assignment merely copies a pointer to the original string. With IString objects, the assignment copies each byte of the string into the new string.
- You do not have to determine the length of the source string and allocate sufficient storage to store it in the target string before the assignment. IString takes care of allocating the storage for you, whether the target string is being constructed within the assignment or has already been constructed. This reduces the risk of memory violations. In the following example, String2 is constructed and initialized, and then copied to (its original contents are overwritten), while String3 is copy-constructed to contain a copy of String1. Notice that String2's length is extended by the assignment operation.

  ```
  IString String1="A longer string than String2";
  IString String2="A short string";
  IString String3=String1;    // initialized to String1
  String2=String1;            // extended to fit String1
  ```

- The string being copied can contain null characters anywhere within it, and the entire string will be copied.
- If you accidentally create an array of characters without the terminating null, the strcpy function may continue copying past the storage allocated for the string. This can cause storage violations, or, at the least, it can corrupt the data in the target string. The length of IString objects is not determined by a terminating null, so storage violations and corrupt target strings are less likely.

Creating Substrings of Strings

You can use the subString function to return a new IString object containing a portion of another IString. This function lets you create an IString containing the leftmost characters, rightmost characters, or characters in the string's middle. The following example shows calls to subString that create substrings with leftmost, rightmost, or middle characters:

```
// Using the subString method of IString

#include <iostream.h>
#include <istring.hpp>

void main() {
   IString All("This is the entire string.");

   // Left -> subString(1, length)
   IString Left=All.subString(1,5);

   // Middle -> (startpos, length)
   IString Middle=All.subString(6,14);

   // Right -> (string length - (substring length - 1) )
   IString Right=All.subString(All.length()-6);

   cout << "<" << All << ">\n"
        << "<" << Left << ">\n"
        << "<" << Middle << ">\n"
        << "<" << Right << ">" << endl;
   }
```

This program produces the following output:

```
<This is the entire string.>
<This >
<is the entire >
<string.>
```

Doing String Input and Output

The `IString` class overloads the input and output operators of the I/O Stream Class Library so that you can extract `IString` objects from streams and insert `IString` objects into them. The input operator reads characters from the input stream until a white-space character or EOF is encountered. The `IString` class also defines a member function to read a single line from an input stream. The following example shows uses of the input and output operators for `IString` and the `lineFrom` function:

```
//Using the IString I/O operators and the lineFrom function

#include <istring.hpp>
#include <iostream.h>

void main() {
    IString Str1, Str2, Str3;
    Str1="Enter some text:";
    char test[80];

    // Write prompt
    cout << Str1;
    // Get input
    cin >> Str2;
    // This only reads in one word of text, so we should
    // check to see if this was the only word on the line:
    if (cin.peek()!='\n') {
       // there's more text on this line so ignore it
       cin.ignore(1000,'\n');
       }
    // Change prompt
    Str1.insert("more ",Str1.indexOf(" text:"));
    // Write prompt again
    cout << Str1;
    // Get line of input
    Str3=IString::lineFrom(cin,'\n');
```

```
   // Write output
   cout << "First word of first input: " << Str2 << '\n'
        << "Full text of second input: " << Str3 << endl;
   }
```

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

```
Enter some text:Here is my first string
Enter some more text:Here is my second string
First word of first input: Here
Full text of second input: Here is my second string
```

Note that, although null characters are allowed within an `IString` object, a null character in an input string is treated as the end of the input, and a null character in an `IString` being written to an output stream ends the output of that `IString`.

Concatenating Strings

The `IString` class defines an addition operator (+) to allow you to concatenate two words together. An addition assignment operator (+=) lets you assign the result of the concatenation to the left operand. The `copy()` member function lets you create an `IString` consisting of multiple copies of itself or of another string. The following example shows ways of concatenating text onto the start or end of an `IString`:

```
// Concatenating strings

#include <iostream.h>
#include <istring.hpp>

void main() {
   IString Str1="Let ";
   IString Str2="us ";
   IString Str3="concatenate ";
   IString Str4="repeatedly ";

   IString Str5=Str1+Str2;  // Add Str1 and Str2 and store in Str5;
   Str5+=Str3;              // Add Str3 to Str5
   Str4.copy(3);            // Copy Str4 several times onto itself
   Str5+=Str4;              // Add Str4 to Str5
   cout << Str5 << endl;    // Write String 5
   }
```

This program produces the following output:

```
 Let us concatenate repeatedly repeatedly repeatedly
```

Finding Words or Substrings within Strings

A wide range of functions are available to let you find words, substrings, patterns, or individual characters within a string. You can even do wildcard searches: for example, you can search through a string to find a substring that begins with the letters `"Ar"` followed by one or more characters, followed by the letters `"rk"`.

The following example shows a number of the searching functions available for `IString` objects. Comments describe the type of search operation being carried out.

```
// Searching for substrings

#include <iostream.h>
#include <istring.hpp>
```

```
void main() {
   IString Str1="This string contains some sample text in English.";
   IString Str2=Str1.subString(27); // positions 27 and following:
                                   // "sample text in English."
   cout << "The string under consideration is:\n\n"
        << Str1 << "\n\n";

// 1. Count the number of occurrences of a substring within the string

   cout << "The substring \"in\" occurs "
        << Str1.occurrencesOf("in")
        << " times in the string.\n";

// 2. Find the first occurrence of a substring:
//    (Note that the substring can be a char, char*, or IString value)

   cout << "The letter 'x' first occurs at position "
        << Str1.indexOf('x') << ".\n";

// 3. Find the first occurrence of any letter of those specified:

   cout << "One of the letters q, r, or s first appears at position "
        << Str1.indexOfAnyOf("qrs") << ".\n";

// 4. Find the first occurrence of any letter other than those specified:

   cout << "The first letter that is not in \"Think\" "
        << "appears at position "
        << Str1.indexOfAnyBut("Think") << ".\n";

// 5. Find the index of a word

   cout << "The third word starts at position "
        << Str1.indexOfWord(3) << ".\n";

// 6. Find a match to a phrase, and return the position of the
//    first matching word

   cout << "The phrase \"" << Str2 << "\" starts at word number "
        << Str1.wordIndexOfPhrase(Str2) << " of the string.\n";

// 7. Do a wildcard search to see if the string starts with "Th",
//    contains "co", and ends with "sh."

   cout << "Does the string match the wildcard search string "
        << "\"Th*co*sh.\"?\n";
   if (Str1.isLike("Th*co*sh.")) cout << "Yes.";
   else cout << "No.";

   cout << endl;
   }
```

This program produces the following output:

```
The string under consideration is:

This string contains some sample text in English.

The substring "in" occurs 3 times in the string.
The letter 'x' first occurs at position 36.
One of the letters q, r, or s first appears at position 4.
The first letter that is not in "Think" appears at position 4.
The third word starts at position 13.
The phrase "sample text in English." starts at word number 5 of the string.
Does the string match the wildcard search string "Th*co*sh."?
Yes.
```

Replacing, Inserting, and Deleting

Replacing, Inserting, and Deleting Substrings

The ability to manipulate the contents of an IString is one of the greatest advantages of the IString class over the traditional method of using string.h functions to manipulate arrays of characters. Consider, for example, a function that perform the following changes on a string. Issues that you need to address when using arrays of characters, but that are handled for you by the IString class, are shown in parentheses:

1. Replace all occurrences of Blue with Yellow (string must be expanded by two characters for each replacement, and text after the replacement must be shifted out).
2. Replace all occurrences of Orange with Pink (string must be shortened by two characters for each replacement).
3. Delete the sixth word of the string. (How are words delimited? By spaces? Carriage returns? Tab characters? What about multiple adjacent whitespace characters?)
4. Insert the word Dark as the fourth word or at the end of the string if the string has fewer than three words. (String must be extended. How are words delimited? Do you add a space before or after the word?).

You can easily handle the above requirements using IString member functions. The sample function fixString() below implements the requirements. Numbered comments correspond to the numbers of the requirements:

```
// Inserting, deleting and replacing substrings

#include <iostream.h>
#include <istring.hpp>

void fixString(IString&);

void main() {
   IString Str1="Light Blue and Green are nice colors. ";
   Str1+="But so are Red and Orange.";
   cout << Str1 << endl;
   fixString(Str1);
   cout << Str1 << endl;
   }

void fixString(IString &myString) {
   myString.change("Blue", "Yellow");   // 1. Change Blue to Yellow
   myString.change("Orange", "Pink");   // 2. Change Orange to Pink
   myString.removeWords(6,1);           // 3. Remove words, starting at word 6,
                                        //    for a total of 1 word.
   int Word4=myString.indexOfWord(4);
   if (Word4>0)                         // 4. Insert "Dark" as fourth word
      myString.insert("Dark ",Word4-1); //    or at end of string if string
   else                                 //    has fewer than 4 words.  The
      myString+=" Dark";                //    insertion occurs 1 byte before
   }                                    //    word 4 (otherwise it inserts
                                        //    in the middle of word 4).
```

This program produces the following output:

```
Light Blue and Green are nice colors. But so are Red and Orange.
Light Yellow and Dark Green are colors. But so are Red and Pink.
```

Determining String Lengths and Word Counts

You can determine not only the length of a string, but the number of words within the string, or the length of a particular word in the string. The length of a string is not affected by any null characters you insert in the middle of the string. (The `strlen` function of `string.h` treats any null character in an array of characters as a terminating null.)

The following descriptions assume that `ThisString` contains the text “This string has five words.”

The `length` and `size` functions both return the length of an `IString`. For example, `ThisString.size()` returns the value 26, as does `ThisString.length()`.

To determine the number of words in a string, use the `numWords` member function. For example, `ThisString.numWords()` returns the value 5.

To determine the length of a particular word, use the `lengthOfWord` member function. For example, `ThisString.lengthOfWord(3)` returns the value 3.

Extending Strings

With arrays of characters, unless you allocate more storage than originally required for a string, you can only extend a string by allocating a new chunk of storage, moving the existing string into the new area, and extending it there.

`IString` objects are automatically extended for you whenever an `IString` operator or function requires the extension. This lets you spend more time coding useful function, and less time trying to track down the source of memory violations or data corruption. You can even use the subscript operator to assign a value to a position beyond the end of the string. The following example, by indexing past the end of `ShortString`, causes the string to be padded with blanks up to position 119, and the letter “a” is added at position 120:

```
IString ShortString="A short string";
ShortString[120]='a';
```

The + and += operators, the assignment operator, and all member functions that change the contents of a string automatically allocate additional storage for the string if that storage is required. This can drastically reduce the amount of string-handling code you need to write.

Converting between Strings and Numeric Data

The `IString` class provides a number of `as...` functions that convert from `IString` objects to numeric types. You can also convert from numeric types to `IString` objects by using the versions of the `IString` constructor that take numeric values as arguments. The following example shows various `IString` functions that convert between strings and numbers:

```
// Conversion between IString and numeric values

#include <iostream.h>
#include <istring.hpp>
```

Base Conversions

```
void main() {
IString NumStr=1.4512356919E1;   // Initialized with a float value
int Integer=NumStr.asInt();      // Convert to integer value
float Float=NumStr.asDouble();   // C++ conversion rules allow asDouble's
                                 // result to be converted to float
double Double=NumStr.asDouble(); // Convert to double value
NumStr=688;                      // Assign another integer value

cout.precision(20);              // Set precision of cout stream
cout <<   "Integer: " << Integer << "\nFloat:   " << Float
     << "\nDouble:  " << Double  << "\nString:  " << NumStr << endl;
}
```

This program produces the following output:

```
Integer: 14
Float:   14.512356758117676
Double:  14.512356919
String:  688
```

You can also change the base notation of `IString` objects containing integer numbers, by using the `d2...` functions, which convert from decimal to binary, hexadecimal, or character representations. Conversion functions are described in the next section.

Converting between Strings and Different Base Notations

You can use the format conversion functions to change the way the data in a string is represented. These functions are overloaded so that each function has two versions. The nonstatic version replaces the value of the string with the converted value. The static version preserves the original string and returns a new string object containing the converted value. For example:

```
aString.c2b();                          // Changes value of aString
IString binaryDigits = c2b( aString );  // Preserves value of aString
```

The conversion functions check the format of the source string to make sure it is compatible with the source format implied by the function name. For example, if you use the `b2d` function to convert a string from binary to decimal, the function first checks that the string contains only the digits '0' and '1'. If it contains any characters other than those allowed by the source type, the format conversion functions always return 0.

The following example shows the use of the conversion functions. If you examine both the example and the output provided below, you can see how to use the functions.

```
// IString conversion functions

#include <istring.hpp>
#include <iostream.h>
enum Bases {Bin, Dec, Hex, Char};
IString Base[4]={"binary", "decimal", "hex", "character"};
IString NumStr;

void Show(int From, int To, IString& Result) {
   cout << NumStr << " in " << Base[From] << " is "
        << Result << " in " << Base[To] << '.' << endl;
   }
```

```
void main() {
   IString NewStr;
   NumStr="122";
      NewStr=IString::d2b(NumStr); Show(Dec,Bin,NewStr);
      NewStr=IString::d2x(NumStr); Show(Dec,Hex,NewStr);
      NewStr=IString::d2c(NumStr); Show(Dec,Char,NewStr);
   NumStr="Hat";
      NewStr=IString::c2b(NumStr); Show(Char,Bin,NewStr);
      NewStr=IString::c2d(NumStr); Show(Char,Dec,NewStr);
      NewStr=IString::c2x(NumStr); Show(Char,Hex,NewStr);
   NumStr="5F";
      NewStr=IString::x2b(NumStr); Show(Hex,Bin,NewStr);
      NewStr=IString::x2d(NumStr); Show(Hex,Dec,NewStr);
      NewStr=IString::x2c(NumStr); Show(Hex,Char,NewStr);
   NumStr="0110100001101001";
      NewStr=IString::b2d(NumStr); Show(Bin,Dec,NewStr);
      NewStr=IString::b2x(NumStr); Show(Bin,Hex,NewStr);
      NewStr=IString::b2c(NumStr); Show(Bin,Char,NewStr);
   }
```

The output from this program resembles the following. Depending on the code page and character set (ASCII or EBCDIC) of the system you are running the program on, the values may vary.

```
122 in decimal is 01111010 in binary.
122 in decimal is 7A in hex.
122 in decimal is : 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 *IBM VisualAge for C++ for AS/400 IBM Open Class Library Reference*. The following example shows how you can use a subset of these functions:

Testing String Characteristics

```
// Evaluating strings using the IString is... methods

#include <istring.hpp>
#include <iostream.h>

void evaluate(IString& StringToTest) {
   if (StringToTest.isPrintable())
      cout << "Evaluating the string " << StringToTest << ":" << endl;
   else
      cout << "Evaluating an unprintable string:" << endl;
   if (StringToTest.isDigits())
      cout << "   Contains only digits 0-9." << endl;
   if (StringToTest.isAlphabetic())
      cout << "   Contains only alphabetic characters." << endl;
   if (StringToTest.isAlphanumeric())
      cout << "   Contains only alphabetic and numeric characters." << endl;
   if (StringToTest.isBinaryDigits())
      cout << "   Contains only zeros and ones." << endl;
   if (StringToTest.isHexDigits())
      cout << "   Contains only hex digits 0-9, a-f, A-F." << endl;
   if (StringToTest.isControl())
      cout << "   Contains only ASCII values 00-1F, 7F." << endl;
   if (StringToTest.isLowerCase())
      cout << "   Contains only lowercase letters a-z." << endl;
   if (StringToTest.isUpperCase())
      cout << "   Contains only uppercase letters a-z." << endl;
   if (StringToTest.isSBCS())
      cout << "   Contains only SBCS characters." << endl;
   }

void main() {
   IString Str[6];
   Str[0]="12345";              // numeric, hexadecimal
   Str[1]="abcde";              // alphabetic, hexadecimal
   Str[2]="10101";              // numeric, binary
   Str[3]="abCde";              // alphabetic, hexadecimal
   Str[4]="xyz12";              // alphanumeric, lowercase
   Str[5]="\x04\x06\x11\x12";   // control, unprintable

   for (int i=1;i<6;i++) evaluate(Str[i]);
   }
```

The output from this program resembles the following. Depending on the code page and character set (ASCII or EBCDIC) of the system you are running the program on, the results may vary.

```
Evaluating the string abcde:
   Contains only alphabetic characters.
   Contains only alphabetic and numeric characters.
   Contains only hex digits 0-9, a-f, A-F.
   Contains only lowercase letters a-z.
   Contains only SBCS characters.
Evaluating the string 10101:
   Contains only digits 0-9.
   Contains only alphabetic and numeric characters.
   Contains only zeros and ones.
   Contains only hex digits 0-9, a-f, A-F.
   Contains only SBCS characters.
Evaluating the string abCde:
   Contains only alphabetic characters.
   Contains only alphabetic and numeric characters.
   Contains only hex digits 0-9, a-f, A-F.
   Contains only SBCS characters.
Evaluating the string xyz12:
   Contains only alphabetic and numeric characters.
   Contains only SBCS characters.
```

```
Evaluating an unprintable string:
   Contains only ASCII values 00-1F, 7F.
   Contains only SBCS characters.
```

Formatting Strings

You can insert padding (white space) into strings so that each string in a group of strings has the same length. The `center`, `leftJustify`, and `rightJustify` functions all do this; their names indicate where they place the existing string relative to the added white space. You provide the final desired length of the string, and the function adds the correct amount of white space (or removes characters if the string is longer than the final length you specify). For example:

```
// Padding IStrings

#include <istring.hpp>
#include <iostream.h>

void main() {
IString s1="Short", s2="Not so short",
        s3="Too long to fit in the desired field length";
s1.rightJustify(20);
s2.center(20);
s3.leftJustify(20);
cout << s1 << '\n' << s2 << '\n' << s3 << endl;
}
```

This program produces the following output:

```
               Short
    Not so short
Too long to fit in t
```

If a string is too wide, you can strip leading or trailing blanks using the `strip...` functions:

```
// Using the strip... functions of IString

#include <istring.hpp>
#include <iostream.h>

void main() {
   IString s1,  s2,  s3,  Long="      Lots of space here     ";
   s1 = s2 = s3 = Long;
   s1.stripLeading();
   s2.stripTrailing();
   s3.strip();
   cout << ">" << Long << "<\n"
        << ">" << s1   << "<\n"
        << ">" << s2   << "<\n"
        << ">" << s3   << "<" << endl;
}
```

This program produces the following output:

```
>      Lots of space here     <
>Lots of space here     <
>      Lots of space here<
>Lots of space here<
```

You can also change the case of an IString to all uppercase or all lowercase:

```
// Changing the case of IStrings

#include <iostream.h>
#include <istring.hpp>

void main() {
   IString Upper="MANY of THESE are UPPERCASE CHARACTERS";
   IString Lower="Many of these ARE lowercase characters";
   Upper.change("MANY","NONE").lowerCase();
   Lower.change("Many","None").upperCase();
   cout << Upper << '\n' << Lower << endl;
   }
```

This program produces the following output:

```
none of these are uppercase characters
NONE OF THESE ARE LOWERCASE CHARACTERS
```

Other IString Capabilities

This section has described only a portion of the functionality of the IString class. Many functions described here are overloaded to provide a wider range of functionality, and many of the functions of the IString class were not described here. See the *IBM VisualAge for C++ for AS/400 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:

```
// 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()**.

IStringTest Class

Chapter 18. Exception and Trace Classes

This chapter outlines some of the ways that you can use the exception and trace classes. The exception classes are a set of classes that allow you to catch exceptions based on their type. The trace class `ITrace` allows you to conveniently put trace statements in your programs.

Introduction to the Exception Classes

There are three primary ways to use the exception classes:

1. Certain functions in IBM class libraries throw exceptions that are objects of the exception classes. If you are familiar with the characteristics of the exception classes, you can take advantage of the exception classes to make your code that uses the IBM class libraries more robust.
2. You can both throw and catch objects of the exception classes in your own code. The exception classes provide a convenient way to package information about an exception.
3. You can derive your own classes from the exception classes.

Characteristics of the Exception Classes

The exception classes have the following characteristics:

- A stack of exception message text strings. These strings allow you to describe the exception in detail.
- An error ID that lets you uniquely identify what error caused the exception.
- A severity code that lets you determine whether the exception can be recovered from or not.
- Information about where the exception was thrown.

The exception classes' member functions allow you to:

- Add information about where the exception was thrown
- Add text to the description of the exception
- Get the error ID of the exception
- Determine if the exception is recoverable
- Log the exception data
- Set the error ID of the exception
- Set the severity of the exception
- Set a trace function

Derivation of the Exception Classes

The exception classes consist of a base class `IException` and a set of derived classes:

- `IAccessError`
- `IAssertionFailure`
- `IDeviceError`
- `IInvalidParameter`

© Copyright IBM Corp. 1995, 1996

- IInvalidRequest
- IResourceExhausted

In addition, IResourceExhausted has the following derived classes:

- IOutOfMemory
- IOutOfSystemResource
- IOutOfWindowResource

Note: VisualAge for C++ for AS/400 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

Catching Exceptions Thrown by Class Library Functions

Under certain circumstances, member functions of the Collection and Data Type and Exception 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 new Operator Throwing an Exception

For example, suppose that you use the `new` operator to create a huge array of integer pointers. If there is not enough memory available to satisfy a particular request for memory, the `new` operator throws an `IOutOfMemory` exception.

In the following piece of code, a single invocation of the `new` operator exhausts all of the memory that is available for allocation. In this code, the catch statement specifies the base class `IException` rather than `IOutOfMemory`. If you know that a member function may throw an exception class object, but you do not know its exact type, you can specify a catch statement like this one to catch all of the possible exception class exceptions.

```
//   The new operator throwing an exception

   #include <iostream.h>
   #include <iexcept.hpp>
   #include <istring.hpp>

   #define TOOBIG 1000000000

   void main() {
      int i;
      try {
         int* istr = new int[TOOBIG];
      }
      catch(IException &ie)
      {
        cout << "Type of exception is: " << ie.name() << endl;
        cout << "Location of exception is: "
             << ie.locationAtIndex(0)->fileName() << endl;
        if (ie.isRecoverable())
           cout << "Exception is recoverable" << endl;
        else
           cout << "Exception is unrecoverable" << endl;
      }
   }
```

Assuming that the constant `TOOBIG` is large enough to exhaust all of the memory available for allocation, this code produces the following output:

```
Type of exception is: IOutOfMemory
Location of exception is: ibase.C
Exception is unrecoverable
```

An Example of the Subscript Operator Throwing an Exception

The subscript operator of the `IString` class can throw exceptions that are objects of the exception classes. If you use the subscript operator on an `IString` object that is declared **const**, the operator will throw an `InvalidRequest` exception if the index is out of the bounds of the `IString` object.

Throwing Your Own Exceptions

In the following piece of code, an `IString` object is declared **const**, and then the subscript operator is used with an index beyond the size of the object.

```
// Example that causes a subscript out of bounds exception

#include <iostream.h>
#include <iexcept.hpp>
#include <istring.hpp>

void main() {
   try {
      const IString ConstStr = "OFF";
      cout << ConstStr[4] << endl;
   }
   catch(IException &ie)
   {
     cout << "Type of exception is: " << ie.name() << endl;
     cout << "Location of exception is: "
          << ie.locationAtIndex(0)->fileName() << endl;
     if (ie.isRecoverable())
        cout << "Exception is recoverable" << endl;
     else
        cout << "Exception is unrecoverable" << endl;
   }
}
```

Because the index is beyond the size of the `IString` object, the subscript operator throws an exception. When this code is run, the following output is produced:

```
Type of exception is: IInvalidRequest
Location of exception is: istring5.C
Exception is recoverable
```

Member functions in the Collections and User Interface class libraries also throw exceptions that are objects of the exception classes. If you call such functions within `try` blocks followed by a `catch` statement for `IException` exceptions, you can:

- Make your code more robust by detecting and dealing with exceptions that occur in class library calls.
- Determine why exceptions are occurring by examining the information that is passed back in the exception class object.

Throwing Your Own Exceptions Using the Exception Classes

In addition to catching exception class exceptions that are thrown by class library functions, you can also throw them in your own code. Throwing exception class exceptions in your own code has the following advantages:

- The exception classes provide a convenient package for exception information.
- If you use one of the predefined exception classes or derive one of your own from `IException`, you can use the same `catch` statement to catch exceptions that are generated by both class library functions and your own functions.

Consider the following simple example. The `getFirstChar` function calls the `IASSERTSTATE` macro with a `get` call as an argument. If the `get` call fails, it returns zero and the `IASSERTSTATE` macro throws an `IInvalidRequest` exception.

```
//   Using the IASSERTSTATE macro

   #include <iostream.h>
   #include <fstream.h>
   #include <iexcept.hpp>

   void openFile(fstream& fs, char *filename){
      fs.open(filename, ios::in);
   }

   char getFirstChar(fstream& fs) {
      char c;
      IASSERTSTATE(fs.get(c));
      return c;
   }

   void main() {
      char c;
      char * filename = "source.dat";
      fstream fs;
      openFile(fs, filename);
      try {
         c = getFirstChar(fs);
         cout << "Here is first character: " << c << endl;
      }
      catch(IException &ie)
      {
         cout << "Type of exception is: " << ie.name() << endl;
         cout << "Location of exception is: "
              << ie.locationAtIndex(0)->fileName() << endl;
         if (ie.isRecoverable())
            cout << "Exception is recoverable" << endl;
         else
            cout << "Exception is unrecoverable" << endl;
      }
   }
```

Suppose that this example is run, and the `source.dat` file is not available. The call to `open` in the `OpenFile` function will fail. When `getFirstChar` is called within the try block, an exception will be thrown by the `IASSERTSTATE` macro. This exception will be caught by the `catch` statement in `main`, and the output will look something like this:

```
Type of exception is: IInvalidRequest
Location of exception is: iopen.C
Exception is recoverable
```

Macros Used with the Exception Classes

The exception classes support a set of macros that allow you to manage the exception classes conveniently. You can use these macros to throw exceptions and to declare and define subclasses of `IException` or one of its subclasses.

`ITHROW`
: Accepts as input an object of any `IException` subclass. It expands to add the location information to the instance, logs all instance data, and then throws the exception.

`IRETHROW`
: Accepts as input a predefined instance of any subclass of `IException` that has been previously thrown and caught. Like the ITHROW macro, it also

captures the location information, and logs all instance data before rethrowing the exception.

`IASSERTSTATE`
: This macro accepts an expression to be tested as input. The expression is *asserted* to be true, meaning that you anticipate that it is true and are stating so to the compiler. If it evaluates to false, it invokes the `IExcept__assertState` function, which creates an `IInvalidRequest` exception. Location information is added to the exception, which is then logged and thrown.

`IASSERTPARM`
: This macro accepts an expression to be tested as input. The expression is asserted to be true. If it evaluates to false, it invokes the `IExcept__assertParameter` function, which creates an `IInvalidParameter` exception. Location information is added to the exception, which is then logged and thrown.

`IEXCLASSDECLARE`
: Creates a declaration for a subclass of `IException` or one of its subclasses.

`IEXCLASSIMPLEMENT`
: Creates a definition for a subclass of `IException` or one of its subclasses.

`IEXCEPTION_LOCATION`
: Expands to create an instance of the `IExceptionLocation` class.

`INO_EXCEPTIONS_SUPPORT`
: Provided in support of compilers that lack exception handling implementation. If it is defined, the ITHROW macro ends the program after capturing the location information and logging it, instead of throwing an exception. This macro may not work correctly on all compilers.

`ITHROWGUIERROR`
: This macro takes as its only argument the name of the GUI function that returned an error code. It calls the `IGUIError::throwGUIError` function, which creates an 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 VisualAge for C++ for AS/400. 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 `IBaseErrorInfo::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 VisualAge for C++ for AS/400. 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 IBaseErrorInfo::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:

```
//   Invoking the IException member functions directly

  #include <iostream.h>
  #include <fstream.h>
  #include <iexcept.hpp>

  void openFile(fstream& fs, char *filename){
     fs.open(filename, ios::in);
  }

  char getFirstChar(fstream& fs) {
     char c;
     if (!fs.get(c)) {
        IInvalidRequest ir(" ", 0, IException::recoverable);
        IExceptionLocation il("imac.C","getFirstChar",5);
        ir.addLocation(il);
        throw(ir);
     }
     return c;
  }

  void main() {
     char c;
     char * filename = "source.dat";
     fstream fs;
     try {
        c = getFirstChar(fs);
        cout << "Here is first character: " << c << endl;
     }
     catch(IException &ie)
     {
        cout << "Type of exception is: " << ie.name() << endl;
        cout << "Location of exception is: "
             << ie.locationAtIndex(0)->fileName() << endl;
        if (ie.isRecoverable())
           cout << "Exception is recoverable" << endl;
        else
           cout << "Exception is unrecoverable" << endl;
     }
  }
```

Notice how the single `IASSERTSTATE` in the `getFirstChar` function is replaced with a test of the return value of `get`, the definition of an `IInvalidRequest` object, the definition of an `IExceptionLocation` object, and an explicit `throw` statement. You can see that the version of the program that uses the `IASSERTSTATE` macro is simpler and easier to code.

Using the ITrace Class

The `ITrace` class provides a set of facilities that allow you to put trace statements in your code conveniently. The most convenient way to use `ITrace` is through the macros that it supports.

Using the Trace Macros to Control Trace Output

The `ITrace` class is convenient to use because it allows you to turn trace statements on and off easily. By defining certain macros and by using the macros in the `ITrace` class to create trace output, you can selectively turn tracing on and off. There are three special trace macros:

- `IC_TRACE_RUNTIME`
- `IC_TRACE_DEVELOP`
- `IC_TRACE_ALL`

By defining or not defining these macros, you can specify whether or not the trace macros are expanded, and thus whether or not your program produces trace output.

If `IC_TRACE_RUNTIME` is defined, the following macros are expanded:

`IMODTRACE_RUNTIME`
: This macro takes one argument that is the name of the current module. It creates an `ITrace` object using the module name as the name of the trace and the current line number as the line number.

`IFUNCTRACE_RUNTIME`
: This macro takes no arguments. It creates an `ITrace` object using the function name as the name of the trace and the current line number as the line number.

`ITRACE_RUNTIME`
: This macro takes a single argument. This argument is written to the trace location.

If `IC_TRACE_DEVELOP` is defined, all of the macros that are expanded when `IC_TRACE_RUNTIME` is defined, are also expanded. In addition, the following macros are expanded:

`IMODTRACE_DEVELOP`
: This macro takes one argument. Typically you use the argument to name the current module. This macro creates an `ITrace` object using the module name as the name of the trace and the current line number as the line number.

`IFUNCTRACE_DEVELOP`
: This macro takes no arguments. It creates an `ITrace` object using the function name as the name of the trace and the current line number as the line number.

`ITRACE_DEVELOP`
: This macro takes a single argument. This argument is written to the trace location.

If `IC_TRACE_ALL` is defined, all of the trace macros are expanded.

Using the ITrace Class

An Example of Using ITrace

The following piece of code shows one way that you could use the trace macros to produce trace output for your programs. In this code, the macros `IFUNCTRACE_DEVELOP` and `ITRACE_DEVELOP` are used to create trace statements that indicate that the flow of control has passed through the functions `openFile` and `getFirstChar`.

```
// Producing trace output with the ITrace class

#define IC_TRACE_DEVELOP

#include <iostream.h>
#include <fstream.h>
#include <iexcept.hpp>
#include <itrace.hpp>

void openFile(fstream& fs, char *filename){
   IFUNCTRACE_DEVELOP();
   fs.open(filename, ios::in);
   ITRACE_DEVELOP("after open statement");
}

char getFirstChar(fstream& fs) {
   char c;
   IFUNCTRACE_DEVELOP();
   fs.get(c);
   ITRACE_DEVELOP("after get statement");
   return c;
}

void main() {
   char c;
   char * filename = "source.dat";
   fstream fs;
   //
   // static functions to enable tracing and direct
   // tracing output to standard output
   //
   ITrace::enableTrace();
   ITrace::writeToStandardOutput();
   openFile(fs, filename);
   c = getFirstChar(fs);
   cout << "Here is first character: " << c << endl;
}
```

Notice that, in this code, the static functions `enableTrace` and `writeToStandardOutput` are used to enable tracing and to direct the trace output to standard output.

Because the macro `IC_TRACE_DEVELOP` is defined, the trace macros produce trace output. In addition, the trace output has been explicitly directed to standard output, so the output of the code looks like this:

```
+openFile(fstream&,char*)
  >after open statement
-openFile(fstream&,char*)
+getFirstChar(fstream&)
  >after get statement
-getFirstChar(fstream&)
Here is first character: t
```

Suppose that you wanted to turn off the trace output in this program. One way to do it is to modify the code so that the macro IC_TRACE_DEVELOP is not defined. If you do this, the trace macros are not expanded, and no trace output is produced. The output of this code with IC_TRACE_DEVELOP not defined looks like this:

```
Here is first character: t
```

Using the ITrace Class

Chapter 19. 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 postions 3 through 5. If the day of the year is less than three digits, zeros are added on the left to increase the size to three digits. For example, February 14, 1965 is 65045 as a Julian date. (February 14 is the 45th day of the year.)

The IDate class returns the names of the days and months in the language defined by the current locale. For information on defining the locale, see the standard C library function setlocale().

Creating an IDate Object

You can create an IDate object using different IDate constructors. For example:

```
IDate OneDay(IDate::June,30,1994);      // Month, day, year
IDate AnotherDay(23,IDate::April,1961); // Day, month, year
IDate SomeDay(940616);                  // Julian date format
IDate Yesterday(1994,177);              // Year, day of year
```

The constructors accepting a month use the IDate enumeration Month, whose members are named January through December (the months of the year in English).

© Copyright IBM Corp. 1995, 1996

Changing an IDate Object

You can add days to, or subtract days from, an `IDate` object. You can also subtract one date from another, in which case the result is the number of days between the two dates. For example:

```
IDate Day1, Day2;
int NumDays;
Day1=IDate::today();
Day2=Day1+1;         // Day2 is one day after Day1
Day2+=2;             // Day2 is now three days after Day1
NumDays=Day2-Day1;   // NumDays=3
```

Note that you cannot add two `IDate` objects together, because such an addition does not make sense. However, you can add two `ITime` objects together.

Information Functions for IDate Objects

The `IDate` class defines information functions that you can use to obtain specifics about an `IDate` object. For example, you can find out what day of the week, month, or year an `IDate` object's date falls on, or what the name of the day or month is for the current locale. You can also find out what today's date is. The following example shows some of the `IDate` information functions:

```
//  Information functions for IDate class

#include <iostream.h>
#include <istring.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 ITime's hour, minute, or second settings; the other information function returns the current time as determined by the system clock. For example:

```
ITime Time1(ITime::now());
cout << Time1.hours() << " o'clock occurred "
     << Time1.minutes() << " minutes and "
     << Time1.seconds() << " seconds ago." << endl;
```

This displays a result such as the following:

```
12 o'clock occurred 16 minutes and 23 seconds ago.
```

Comparing ITime Objects

Functions are defined to let you compare ITime objects for equality, inequality, or relative position in time. The following operators are defined: ==, !=, <, <=, >, >=. In the following example, a message is displayed if enough time elapses between the first and second calls to the now() member function:

```
#include <itime.hpp>
#include <iostream.h>
ITime First(ITime::now());
void main() {
   ITime Second=ITime::now();
   if (First<Second) // Some time has passed
      cout << "You must be debugging me!" << endl;
   }
```

This message usually does not print when the program is run outside of a debugging session. However, if you debug the program and step through each line slowly, the message may be displayed, because the first ITime object is initialized during program initialization (before **main** is called) while the second ITime object is initialized within **main**.

Writing an ITime Object to an Output Stream

ITime defines an output operator that writes an ITime object to an output stream in the format hh:mm:ss. If you want to write the object out in a different format, you should convert the object to an IString using the asString member function. This member function accepts a char* argument containing a format specifier. The format specifier is the same one as used by the C library function **strftime**. The following program displays some valid specifiers and the output they produce:

```
// Examples of ITime output

#include <istring.hpp>
#include <itime.hpp>
#include <iostream.h>
#include <iomanip.h>  // needed for setw(), to set output stream width

void main() {
   char* FormatStrings[]={
      "%H : %M and %S seconds", // %H, %M, %S - 2 digits for hrs/mins/secs
      "%r",                     // %r - standard 12-hour clock with am/pm
      "%T",                     // %T - standard 24 hour clock
      "%T %Z",                  // %Z - local time zone code
      "%1M past %1I %p"         // %1... - One digit for hour/minute
      };                        // %p - am/pm

   cout.setf(ios::left,ios::adjustfield);    // Left-justify output

   cout << setw(30) << "Format String"       // Title text
        << setw(40) << "Formatted ITime object" << endl;

   for (int i=0;i<5;i++) {                   // Show each time
      IString Formatted=ITime::now().asString(FormatStrings[i]);
      cout << setw(30) << FormatStrings[i]
           << setw(40) << Formatted << endl;
      }
   }
```

The program produces output that looks like the following:

```
Format String            Formatted ITime object
%H : %M and %S seconds   16 : 13 and 04 seconds
%r                       04:13:04 PM
%T                       16:13:04
%T %Z                    16:13:04 EST
%1M past %1I %p          13 past 4 PM
```

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 TmStam5(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:

```
ITimeStamp RefDate;
ITimeStamp TmStamp = ITimeStamp::currentTimeStamp();

IDate ADate = TmStamp;
ITime ATime = TmStamp;

cout << TmStamp << " is " << Seconds << " seconds apart from" << endl;
cout << RefDate << endl;
cout << ATime.hours() << ":" << ATime.minutes() << ":";
cout << ATime.seconds90 << "," << 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 =     12345.54321000000000000000
TmStamp2 =     12345.54320999999800000000
```

Chapter 20. 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 visual and nonvisual parts.

Note: Event and attribute notification for visual parts is not available on OS/400 and OS/390. The IWindow class is only available on operating systems supporting a Graphical User Interface.

The notification framework is different from the existing event handler framework. Handlers are capable of stopping the dispatching of events to the remaining handlers in the chain and only work on IWindow objects. This is unsatisfactory for a notification framework, where registered observer objects must always be notified of an event regardless of how the event was handled.

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

Not all User Interface Class Library notifications occur within this framework. The various I*NotifyHandler classes are provided for the visual classes and controls the various WM_CONTROL messages that the operating system controls send back as the user interacts with these controls. Changes that do not result in a callback from the operating system, such as styles, are handled by the User Interface Class Library controls. The controls issue the notifications for these changes.

For examples of how the notification framework can be implemented refer to the chapter on coding notification frameworks in *Building VisualAge C++ Parts for Fun and Profit*.

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 IStaticText::backgroundColor. 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, the size of a shipment, or the label of a push button.

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 Data Type and Exception 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:

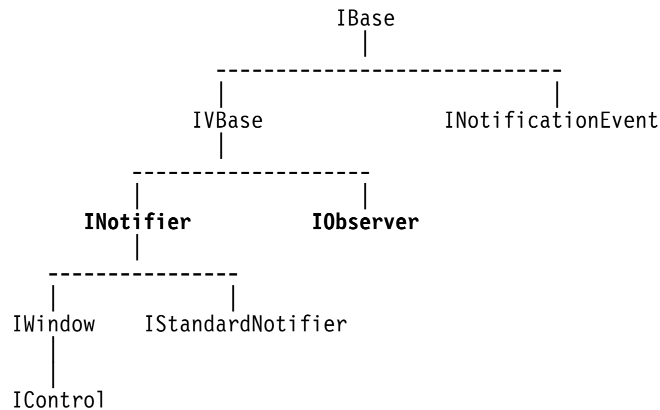
```
  notifyObservers(const INotificationEvent&)
```

  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. (In the window handler framework today, data cannot be stored in a handler object because the handler might be handling many windows.)

The IStandardNotifier class provides the concrete implementation of the notifier protocol and provides the base support for nonvisual notifiers. The notifier protocol is also supported for visual notifiers in the subclasses of IWindow. These classes inherit 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 and IWindow classes are concrete implementations of the notifier protocol.
- Nonvisual notifiers would normally be derived from IStandardNotifier.
- Visual notifiers would normally be derived from IWindow or IControl.

Part 5. The Binary Coded Decimal Class Library for OS/400

This part describes the Binary Coded Decimal Class Library for OS/400 which you can use to create binary coded decimal objects that are compatible with the packed decimal data types in the AS/400 ILE languages.

Chapter 21. Using the Binary Coded Decimal Classes 237
Introducing _DecimalT Class Template 237
Header File and Constants 237
Constructing _DecimalT Template Class Objects 238
__D Macro 238
_DecimalT Class Template Input and Output 239
Mathematical Operators for _DecimalT Template Class 239
Conversions 241
Size of a _DecimalT Template Class 244
Number of Digits of a _DecimalT Template Class 244
Precision of a _DecimalT Template Class 244
_DecimalT Class Template Exceptions 245
Passing a _DecimalT Template Class Object to a Function 249
Passing a Pointer to a _DecimalT Template Class Object 249
Calling Another Program Containing a _DecimalT Template Class 250
Writing a _DecimalT Template Class Constants to a File 251

© Copyright IBM Corp. 1995, 1996

Chapter 21. Using the Binary Coded Decimal Classes

This chapter describes the Binary Coded Decimal Class Library for OS/400. Within this class library there is the `_DecimalT` class template, the `_ConvertDecimal` class, and the `_DecErr` class. With the `_DecimalT` class template you can represent numerical quantities compatible with the AS/400 packed decimal data type. With the `_ConvertDecimal` class you convert one `_DecimalT` template class to another, and convert a large numerical quantity (up to 31 digits) represented in a string literal to a `_DecimalT` class format. With the `_DecErr` class you can identify an exception at runtime.

Note: Use the Binary Coded Decimal Classes if you want your binary coded decimal objects to be compatible with the packed decimal data types in the AS/400 ILE languages.

Introducing _DecimalT Class Template

The `_DecimalT` class template 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 `_DecimalT` class template can store.

The same declarations and operators that you use on other data types, such as `float`, can be applied to `_DecimalT` class templates, with the exception of unions and bitwise operators that do not apply to `_DecimalT` class templates. You can declare typedefs, arrays, and structures that have `_DecimalT` class templates. You can apply arithmetic, relational, assignment, comma, conditional, equality, logical, primary, and unary operators on the `_DecimalT` class template.

You can pass `_DecimalT` template classes in function calls. The `_DecimalT` class template is compatible with packed decimal representations in ILE languages. You can define macros, and call library functions with `_DecimalT` template classes.

Note: You can view the `_DecimalT` template class when you use the ILE system debugger or the cooperative debugger. When you view a `_DecimalT` template class, none of the operators are accessible from the debugger. The *IBM VisualAge for C++ for AS/400 C++ User's Guide* contains information on the ILE system debugger and the VisualAge for C++ for AS/400 cooperative debugger.

Header File and Constants

The class and function template definitions for the `_DecimalT` class template and the numerical limits of a `_DecimalT` template class are defined inside the header file `<bcd.h>`.

You must include this statement in any file that uses the `_DecimalT` class template:

© Copyright IBM Corp. 1995, 1996

_D Macro

```
#include <bcd.h>
```

The file must be included before any use of the `_DecimalT` class template.

Constants Defined in bcd.h

Table 6 lists the constants defined in `<bch.h>`:

Table 6. Constants Defined in bcd.h

Constant Name	Description
DEC_DIG	The maximum number of significant digits that `_DecimalT` class template can hold.
DEC_PRECISION	The maximum number of decimal places that `_DecimalT` class template can hold.
DEC_INT_DIG	The number of significant digits of a binary coded decimal object when you convert it to an integer type. The value 10 is stored in DEC_INT_DIG.
DEC_INT_PREC	The number of decimal places of a binary coded decimal object when you convert it from an integer type. The value 0 is stored in DEC_INT_PREC.

Constructing _DecimalT Template Class Objects

To declare an object as a `_DecimalT` template class:

```
_DecimalT<10,2>  x;

_DecimalT<5,0>   z;
_DecimalT<18,10> *ptr.;
_DecimalT<8,2>   arr[100];
```

The variable `x` can have values between `__D("-99999999.99")` to `__D("+99999999.99");` `z` can have values between `__D("-99999")` to `__D("+99999");` `ptr` is a pointer to class `_DecimalT<18,10>` and `arr` is an array of 100 elements, where each element is of class `_DecimalT<8,2>`.

Note: The string must contain only the characters 0-9, +, . and -.

__D Macro

Use the `__D` macro to simplify code that requires the frequent use of the `_ConvertDecimal` constructor:

```
_ConvertDecimal(char *);
```

You can initialize a `_DecimalT` template class with a `_ConvertDecimal` object:

```
_DecimalT<5,2> x = __D("123.45");
_DecimalT<DEC_DIG, DEC_PRECISION> a =__D(".0000000000000000000000000000005");
_DecimalT<6,2> b[] ={__D("1.2"), __D("2"), __D("1234.56"), _D("-3.3")};
_DecimalT<28,20> a = __D("-12.123456789");
```

_DecimalT Class Template Input and Output

Use `cout` to print the value of a `_DecimalT` template class, and use `cin` to read the value of a `_DecimalT` template class.

To print the value of a `_DecimalT` template class use the `fprintf()`, `printf()`, `sprintf()`, `vfprintf()`, `vprintf()` and, `vsprintf()` functions. To read the value of a `_DecimalT` template class use the `fscanf()`, `scanf()`, and `sscanf()` functions. See the *IBM VisualAge for C++ for AS/400 C Library Reference* for information on the C I/O functions and “Writing a _DecimalT Template Class Constants to a File” on page 251.

Mathematical Operators for _DecimalT Template Class

You can use several operators on `_DecimalT` template classes. You can apply arithmetic, relational, assignment, comma, conditional, equality, logical, primary, and unary operators on the `_DecimalT` class template.

Addition, Subtraction, Multiplication, and Division Operators

To add, subtract, multiply, and divide `_DecimalT` template classes:

```
#include <bcd.h>           // bcd Class Header File
#include <iostream.h>

main()
{
 _DecimalT<10,2> op_1 = __D("12");
 _DecimalT<5,5> op_2 = __D("-.12345");
 _DecimalT<24,12> op_3 = __D("12.34");
 _DecimalT<20,5> op_4 = __D("11.01");

   _DecimalT<14,5> res_add;
   _DecimalT<25,2> res_sub;
   _DecimalT<15,7> res_mul;
   _DecimalT<31,14> res_div;

 res_add = op_1 + op_2;
 res_sub = op_3 - op_1;
 res_mul = op_2 * op_1;
 res_div = op_3 / op_4;

 cout <<"res_add =" <<res_add <<endl;
 cout <<"res_sub =" <<res_sub <<endl;
 cout <<"res_mul =" <<res_mul <<endl;
 cout <<"res_div =" <<res_div <<endl;

 }
```

The output is:

```
res_add =11.87655
res_sub =0.34
res_mul =-1.4814000
res_dev =1.12079927338782
```

Mathematical Operators

Relational Operators

You can use the relational expression “less than,” (<) for _DecimalT template classes and compare _DecimalT template classes with other arithmetic types (integer, float, double, and long double). Implicit conversions are performed using the arithmetic conversion rules. Leading zeros show the size of the number of digits in the _DecimalT template classes. You do not need to enter leading zeros in your _DecimalT template classes:

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

_DecimalT<10,3> pdval = __D("0000023.423"); // bcd declaration
int ival = 1233.1;                          // Integer declaration
float fval = 1234.34f;                       // Float declaration
double dval = 251.5832;                    // Double declaration
long double lval = 37486.234;              // Long double declaration

main( )
{
  _DecimalT<15,6> value = __D("000485860.085999");
  // Perform relational operation between other data types and
  // bcd class

  if (pdval < ival) cout <<"pdval is the smallest !"<<endl;
  if (pdval < fval) cout <<"pdval is the smallest !"<<endl;
  if (pdval < dval) cout <<"pdval is the smallest !"<<endl;
  if (pdval < lval) cout <<"pdval is the smallest !"<<endl;
  if (pdval < value) cout <<"pdval is the smallest !"<<endl;
}
```

The output is:

```
pdval is the smallest !
pdval is the smallest !
pdval is the smallest !
pdval is the smallest !
pdval is the smallest !
```

Equality Operators

You can use equality operators with _DecimalT template classes to initialize and compare _DecimalT template classes for equality:

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

_DecimalT<1, 0>    op_1 = __D("+0");      // Declare and initialize
_DecimalT<1, 0>    op_2 = __D("-0");      // valid BCD
_DecimalT<9,4>     op_3 = __D("00012.3400");
_DecimalT<4,2>     op_4 = __D("12.34");

main(void)                            // These statements
{                                      // perform equality <==> test
                                  // on the above variable
  if (op_1 == op_2)                 // declarations
  {
      cout <<"op_1 equals op_2"<<endl;
  }

  if (op_3 != op_4)
  {
      cout <<"op_3 not equals op_4"<<endl;
  }
  else
```

```
  {
      cout <<"op_3 equals op_4"<<endl;
  }
}
```

The output is:

```
op_1 equals op_2
op_3 equals op_4
```

Conditional Expressions

You can use conditional expressions with `_DecimalT` template classes:

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

main (void)
{
_DecimalT<10,2> x, y, z;
x = __D("1.20");
y = __D("01.2");
z = (x==y)? __D("9.9"):__D("2.45");
if (z== __D("9.9"))
  {
  cout <<"x equals y" <<endl;
  }
}
```

Conversions

The compiler uses different arithmetic conversion rules to convert the operand for all operators prior to the operation, when the operands are not of the same type. This automatic conversion is known as **implicit conversion**.

The implicit conversions are performed using the arithmetic conversion rules.

The compiler implicitly converts these types:

- `_DecimalT` template class to a `_DecimalT` template class
- `_DecimalT` template class to and from a `float` type
- `_DecimalT` template class to and from an `integer` type

An **explicit conversion** is one in which conversion results from a cast operation.

You can explicitly cast:

- `_DecimalT` template class to and from a `float double` or `long double` type
- `_DecimalT` template class to and from an `integer` type

To explicitly cast a `_DecimalT` template class to a `float` type and to an `integer` type:

```
#include <bcd.h>
#include <stdio.h>

main (void)
{
    _DecimalT<10,2> op_1 = __D("-1123.4");
    _DecimalT<12,5> op_2 = __D("12");
```

```
    _DecimalT<4,0> op_3 = __D("5");
    printf("op_1 = %D(5, 1)\n", (_DecimalT<5,1>) __D(op_1));
    printf("op_2 = %f\n", (float) op_2);
    printf("op_3 = %d\n", (int) op_3);
}
```

The output is:

```
op_1 = -1123.4
op_2 = 12.000000
op_3 = 5
```

Note: You cannot explicitly cast a `_DecimalT` template class to a `_DecimalT` template class. See the portability information in the *VisualAge for C++ for AS/400 C++ Programming Guide* for an example.

_DecimalT Template Class to a _DecimalT Template Class

If the value of a `_DecimalT` template class that is to be converted to another `_DecimalT` template class is not within the range of values that can be represented exactly, the value of the `_DecimalT` template class to be converted is truncated. If truncation occurs in the fractional part, there is no run-time exception. If assignment causes truncation in the integral part, then there is a run-time exception in which a `_DecErrDigTruncated` object is thrown. This run-time exception occurs when an integral value is lost during conversion to a different type, regardless of what operation requires the conversion:

```
_DecimalT<4,2> targ_1, targ_2;
_DecimalT<6,2> op_1=__D("1234.56"), op_2=__D("12.34");

targ_1=op_1; // A run-time exception is generated because the integral
               // part is truncated; targ_1=__D("34.56").

targ_2=op_2;  // No run-time exception is generated because neither the
               // integral nor the fractional part is truncated;
               // targ_2=__D("12.34").
```

A run-time exception occurs on assignment to a smaller target only when the integral part is truncated. See “_DecimalT Class Template Exceptions” on page 245 for information on run-time exceptions during conversion.

When one `_DecimalT` template class is assigned to another `_DecimalT` template class with a smaller precision, the result is truncation of the fractional part.

```
_DecimalT<7,4> x = __D("123.4567");
_DecimalT<7,1> y;

y = x;    // y = __D("123.4")
```

When one `_DecimalT` template class is assigned to another `_DecimalT` template class with a smaller integral part, the result is truncation of the integral part. A run-time exception occurs. See “_DecimalT Class Template Exceptions” on page 245 for information on run-time exceptions during conversion.

```
_DecimalT<8,2> x = __D("123456.78");
_DecimalT<5,2> y;

y = x;   // y = __D("456.78")
```

Conversions

When one _DecimalT template class is assigned to another _DecimalT template class with a smaller integral part, and smaller precision, the result is truncation of the integral, and fractional parts. A run-time exception occurs. See "_DecimalT Class Template Exceptions" on page 245 for information on run-time exceptions during conversion.

```
_DecimalT<8,2> x = __D("123456.78");
_DecimalT<4,1> y;

y = x;  // y = __D("456.7")
```

A _DecimalT Template Class to an Integer Type

When a value of a _DecimalT template class is converted to an integer type, the value is first converted to _DecimalT<10,0>, and then this _DecimalT template class is converted to an integer type. No run-time exception occurs, when assigning a _DecimalT template class to an integer type that results in truncation of the integral part.

To convert an integer type to a _DecimalT template class with a fractional part:

```
int op;
_DecimalT<7,2> op1 = __D("12345.67");
op = op1;                         // Truncation on the fractional
                              // part. op=12345
```

To convert an integer type to a _DecimalT template class with less than 10 digits in the integral part:

```
int op;
_DecimalT<3, 0> op2 = __D("123");
op = op2;       // No truncation; op=123
```

To convert to an integer type from a _DecimalT template class with more than 10 digits in the integral part:

```
int op2;
_DecimalT<12, 0> op3;
op3 = __D("123456789012");
op2 = op3;                // Truncation occurs on the integral
                     // part. op2=3456789012; no runtime
                     // exception.
```

To convert to an integer type from a _DecimalT template class with a fractional part, and with an integral part that has more than 10 digits:

```
#include <bcd.h>

int op;
_DecimalT<15,2> op_1 = __D("1234567890123.12");
op = op_1;               // Truncation occurs on the integral and
                    // fractional parts. op=4567890123; no
                    // run-time exception.
```

A _DecimalT Template Class to a Floating Point Type

To convert a _DecimalT template class to a floating-point type:

Precision

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

int main(void)
{
  _DecimalT<5,2> dec_1=__D("123.45");
  _DecimalT<11,5> dec_2=__D("-123456.12345");

  float f1,f2;

  f1=dec_1;
  f2=dec_2;

  cout <<"f1=" <<f1 <<endl <<"f2=" <<f2 <<endl <<endl;//f1=123.45
                                               // f2=-123456
}
```

The output is

```
f1=123.45
f2=-123456
```

Size of a _DecimalT Template Class

When you use the `sizeof` operator with `_DecimalT<n, p>`, you can find out the total number of bytes occupied by the `_DecimalT` template class.

Note: Each `_DecimalT` template-class digit occupies half a byte. Half a byte is used for the sign. The number of bytes used by `_DecimalT<n,p>` is the smallest whole number greater than or equal to `(n + 1)/2` (for example, `sizeof(_DecimalT<n, p>) = ceil((n + 1)/2)`.

To determine the total number of bytes occupied by a `_DecimalT` template class:

```
int y;
_DecimalT <5, 2> x;
                   // This would be calculated to be 3 bytes
 y =sizeof(x);     // (5+1)/2 = 3.
```

Number of Digits of a _DecimalT Template Class

When you use the member function `DigitsOf()` with a `_DecimalT` template class, you can find out the total number of digits `n` in a `_DecimalT` template class.

To determine the number of digits `n` in a `_DecimalT` template class:

```
int n,n1;
_DecimalT <5, 2> x;
n = x.DigitsOf();                // the result is n=5
```

Precision of a _DecimalT Template Class

When you use the member function `PrecisionOf()` with a `_DecimalT` template class, you can find out the number of decimal digits `p` in a `_DecimalT` template class.

To determine the number of decimal digits p of the _DecimalT template class:.

```
int p,p1;
_DecimalT <5, 2> x;
p=x.PrecisionOf();                // The result is p=2
```

Overflow Behavior

Table 7 describes the overflow behavior when a _DecimalT template class is assigned to a smaller target.

An exception is generated when a _DecimalT template class is assigned to a _Decimal template class with a smaller target.

An exception is not generated when a:

- _DecimalT template class is assigned to a char, int, short, long, or bit field with a smaller target
- _DecimalT template class is assigned to a floating point with a smaller target

Table 7. Handling Overflow From a _DecimalT Template Class to a Smaller Target

From Type	To Type	Run-time Exception
_DecimalT template class	char, int, short, long, bit	No
_DecimalT template class	_DecimalT template class	Yes
_DecimalT template class	float	No[1]

Note: [1] There is no _DecimalT template class large enough to cause overflow when the _DecimalT template class is assigned to a float.

_DecimalT Class Template Exceptions

This section describes run-time exceptions for _DecimalT template classes, error classes, and debug macros when C++ exception handling is used with the _DecimalT class template. If an error is detected during runtime, an error object is thrown.

Understanding _DecimalT Class Template Run-time Exceptions

In ILE C, a packed decimal is implemented as a native data type. This approach allows an error, such as a decimal format that is not valid, to be detected at compile time. In VisualAge for C++ for AS/400, detection of a similar error is deferred until runtime:

```
#include <bcd.h>
void main()
{
   _DecimalT<10,20> b = __D("ABC"); // Run-time exception is raised

}
```

```
#include <bcd.h>
void main()
{
   _DecimalT<33,2> a;              // Max. dig. allow is 31. Again,
                                   // run-time exception is raised
}
```

Exceptions

Run-time exceptions may occur during these operations: assignment, casting, initialization, arithmetic operators, and function calls. Overflow situations that occur during compilation are deferred until runtime; loss of digits may occur.

The run-time exceptions issued by the compiler for `_DecimalT` template classes are:

```
#include <bcd.h>

static _DecimalT<5,2>   s1 =   __D("12345678.0");

static _DecimalT<10,2>  s2 =   __D("1234567891234567891234567.12345")
                                              + __D("12345.1234567891");

                                          // s2 = (31,5) + (15,10)

static _DecimalT<10,2>  s3 =  __D("1234567891234562345")
                                              * __D("1234567891234.12");

                                          // s3 = (19,0) * (15,2)

static _DecimalT<10,2>  s4 =  __D("12345678912345678912") /
                                           __D("12345.123456789123456");

                                          // s4 = (20,0) / (20,15)

int main(void)

{
    _DecimalT<5,2> a1 = __D("12345678.0");

    _DecimalT<10,2> a2, a3, a4;

    _DecimalT<5,2> a5 = (_DecimalT<5,2>) __D("123456.78");



    a2 = __D("1234567891234567891234567.12345") + __D("12345.1234567891");

                                          // a2 = (31,5) + (15,10)

    a3 = __D("123456789123456.12345") * __D("1234567891234.12");

                                          // a3 = (20,5) * (15,2)
                                          // expression.
                                          // Note: Need (35,7) but
                                          // use (31,2), for example,
                                          // keep the integral part.

    a4 = __D("12345678912345678912") / __D("12345.123456789123456");

                                          // a4 = (20,0) / (20,15)
                                          // Note: Need 35 digits to
                                          // calculate integral part
                                          // and the result becomes
                                          // (31,0).
}
```

Since `_DecimalT` is implemented as a class template, there are no warnings or errors at compile time. All exceptions are deferred to runtime.

Notes:

1. For assignments to a target field too small to hold the `_DecimalT` template class object, there is a run-time exception issued for static, external, or automatic initialization.
2. For expressions requiring decimal-point alignment, namely addition, subtraction, or comparison, there is a run-time exception issued for static, external, and automatic initialization if, during alignment, the maximum number of allowed digits is exceeded.
3. For multiplication, if the evaluation of the expression results in a value larger than the maximum number of allowed digits (`DEC_DIG`):
   - A run-time exception is issued for static or external initialization, or if the expression is within a function
   - Truncation occurs on the fractional part, preserving as much of the integral part as possible.
4. For division, a run-time exception is generated when $((n_1 - p_1) + p_2) > 31$.

Using a Class Derived from the _DecErr Class

An object instantiated from an error class is thrown if an error occurred during runtime. You can catch the exception by defining your own exception handler.

To handle a `_DecimalT` class-template exception:

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

void main()
{
   try
   {
      _DecimalT<10,2> = __D("AAA");
   }
   catch (_DecErrInvalidConst &Err)
   {
      cout <<"Invalid Decimal constant!"<<endl;
   }
}
```

Using Debug Macros

C++ exception handling is used in the `_DecimalT` class template. If an error is detected during runtime, an error object is thrown.

Table 8 on page 248 defines the three macros that are available to you for turning assertion-checking on and off.

Table 8. Debug Macros for _DecimalT Template Classes

Macro	Meaning
_DEBUG	Assertion checking is on
_DEBUG_DECIMAL	Assertion checking is on for the _DecimalT class template only
_NODEBUG_DECIMAL	Assertion checking is off. This macro can override the _DEBUG and _DEBUG_DECIMAL macros.

When assertion checking is on, the _DecimalT template-class constant and parameters of the _DecimalT template class are validated. The default is no assertion checking. Checking for divide-by-zero, overflow, and truncation in the _DecimalT template-class digit is hard-coded in the C++ runtime and cannot be turned off by the _NODEBUG_DECIMAL macro.

You can use the _DEBUG macro to turn assertion checking on for the _DecimalT template class. If you are already using the _DEBUG macro in your source, you can use the _DEBUG_DECIMAL macro to turn on assertion checking, for the DecimalT template class only.

To enable error checking within the _DecimalT template class, you can turn on the debug macro by adding either the -D_DEBUG_DECIMAL macro or the -D_DEBUG macro during the invocation of the compiler:

```
iccas -D_DEBUG temp.cpp
iccas -D_DEBUG_DECIMAL temp.cpp
```

The difference between the first and second invocations depends on whether or not the _DEBUG macro is used by other classes to control the error checking. If the _DEBUG macro is used by another class, the -D_DEBUG macro affects all classes that use the _DEBUG macro and the -D_DEBUG_DECIMAL macro affects only the _DecimalT template class.

You can turn assertion checking on for a group of files using the _DEBUG macro, but use the _NODEBUG_DECIMAL macro to override the _DEBUG macro and turn assertion checking off.

To disable error checking within the _DecimalT class template, you can use this command:

```
iccas -D_NODEBUG_DECIMAL temp.cpp
```

This is the default. The command `iccas temp.cpp` disables error checking.

You can enable error checking for all classes except, the _DecimalT class template, that use _DEBUG as the control macro:

```
iccas -D_DEBUG -D_NODEBUG_DECIMAL temp.cpp
```

Note: If you try to use both the _DEBUG_DECIMAL and _NODEBUG_DECIMAL macros on the same invocation, the _NODEBUG_DECIMAL macro takes precedence, and error checking for the _DecimalT class template is disabled.

Passing a _DecimalT Template Class Object to a Function

There are no default argument promotions on arguments that have _DecimalT class templates when the called function does not include a prototype. Any function definition containing _DecimalT class template arguments must be prototyped; otherwise, the compiler issues an error.

To pass a _DecimalT template class to a function:

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

_DecimalT<3,1>   d1 = __D("33.3");
_DecimalT<10,5>  d2 = __D("55555.55555");
_DecimalT<28,0>  d3 = __D("8888888888888888888888888888");

void func1( _DecimalT<3,1>, _DecimalT<10,5>,
            _DecimalT<10,5>, _DecimalT<28,0>);

int main(void) {

func1(d1, d2, __D("999.99"), d3);

{
// func1 is prototyped
void func1(_DecimalT<3,1> x1, _DecimalT<10,5> x2,
           _DecimalT<10,5> x3, _DecimalT<28,0> x4) {
// no runtime error when referencing x1, x2, x3 or x4
cout <<"x1 = " <<x1 <<endl;
cout <<"x2 = " <<x2 <<endl;
cout <<"x3 = " <<x3 <<endl;
cout <<"x4 = " <<x4 <<endl;
}
```

The output is:

```
x1 = 33.3
x2 = 55555.55555
x3 = 999.99000
x4 = 8888888888888888888888888888
```

Passing a Pointer to a _DecimalT Template Class Object

To pass a pointer to a _DecimalT template class object to a function:

```
// This program shows how to pass a pointer to a _DecimalT template
// class object to a function.

#include <bcd.h>
#include <iostream.h>

  _DecimalT<5,2> var=__D("123.45");
  _DecimalT<5,2> *p=&var;

  _DecimalT<5,2> *func_1(_DecimalT<5,2> *);

int main(void)
{

//  Call function with pointer to _DecimalT template class argument.
//  The value that it returns is a pointer to a _DecimalT
//  template class object.
```

```
    _DecimalT<5,2> *tempvar;
    tempvar = func_1(p);

    if(tempvar == 0)
    {
       cout <<"Function call not successful" <<endl <<endl;
    }
    else
    {
       cout <<"The bcd value is " << *tempvar <<endl;
    }
}

_DecimalT<5,2> *func_1(_DecimalT<5,2> *q)
 {
 return q;
 }
```

The output is:

```
The bcd value is 123.45
```

The `_DecimalT` template-class argument in the function call has to be the same class template as the `_DecimalT` class template in the function prototype. If overflow occurs in a function call with `_DecimalT` template class arguments, a run-time exception is generated.

Calling Another Program Containing a _DecimalT Template Class

You can make interlanguage calls and pass `_DecimalT` template-class arguments to ILE RPG, ILE COBOL, and ILE C programs.

To call an ILE COBOL program from an ILE C++ program and pass a `_DecimalT` template class:

```
// This program calls an ILE COBOL program
// and passes a bcd object.

#include<iostream.h>
#include <bcd.h>

extern "COBOL" void CBLPGM(_DecimalT<9,7>);


int main(void)
{
 _DecimalT<9,7> arg=__D("12.1234567");

//  Call an ILE COBOL/400 program and pass a bcd object
//  to it.

 CBLPGM(arg);

 cout <<"The COBOL program was called and passed a bcd object"<<endl;

 }
```

The ILE COBOL/400 program is:

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID. CBLPGM.
*************************************************************************
*                                                                       *
*   Packed decimals:   This is going to be called by a C++            *
*   program to pass packed decimal data.                                *
*                                                                       *
*************************************************************************
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.

 SOURCE-COMPUTER. IBM-AS400.
 OBJECT-COMPUTER. IBM-AS400.

 INPUT-OUTPUT SECTION.
 FILE-CONTROL.

 DATA DIVISION.

 FILE SECTION.

 WORKING-STORAGE SECTION.

 77  PAC-DATA                    PIC X(30)
                                 VALUE "PROGRAM START".
 77  PACK-IN-WS                  PIC 99.9999999.

 LINKAGE SECTION.

 01  PACK-DATA                   PIC 9(2)V9(7) PACKED-DECIMAL.

 PROCEDURE DIVISION USING PACK-DATA.

 MAIN-LINE SECTION.

     MOVE PACK-DATA TO PACK-IN-WS.
     DISPLAY "****  BCD OBJECT RECEIVED IS: " PACK-IN-WS.
     GOBACK.
```

The output is:

```
**** BCD OBJECT RECEIVED IS: 12.1234567
The COBOL program was called and passed a bcd object
```

Writing a _DecimalT Template Class Constants to a File

To write `_DecimalT` template-class constants to a file, scan them back, and pass a `_DecimalT` template-class array to a function:

```
// This program shows how to write _DecimalT template class
// constants to a file
// and scan them back again. Shows how to pass a _DecimalT
// template class array to a function.

#include <bcd.h>
#include <iostream.h>
#include <stdlib.h>

#define N    3          // Array size for decimal declaration.

FILE *stream;                        // File pointer declaration.

                                    // Declare valid array.
```

Writing _DecimalT Template Class Constants

```
_DecimalT<4,2> arr_1[] = {__D("12.35"), __D("25.00"),
                                    __D("-19.58")};
_DecimalT<4,2> arr_2[N];

void write_num(_DecimalT<4,2> a[N]); //Declare function to
                            // write to a file.

void read_num(_DecimalT<4,2> b[N]); //Declare function to
                             //read from a file.

int main(void)
{
   int reposition=0;
                                  // Open the file. Must use fopen()
                                  // to access a physical file.
   if ((stream = fopen("*CURLIB/OUTFILE","w+")) == NULL)
     {
       cout <<"Can not open file";
       exit(EXIT_FAILURE);
     }
   write_num(arr_1);     // Call function to write values of the
                         // array to outfile with fprintf().

   reposition=fseek(stream, 0L, SEEK_SET);

   if (reposition!=0)
     {
       cout <<"FSEEK failed to position file pointer" <<endl;
       exit(EXIT_FAILURE);
     }

   read_num(arr_2);      // Call function to read values of the
                         // array from file using fscanf().

   fclose(stream);                    // Close the file.

}

// write_num is passed a the array.  These values are written to a
// text file with fprintf(). If the function is successful a 0 is
// returned, otherwise a negative value is returned (indicating an
// error.

void write_num(_DecimalT<4,2> a[N])

{
   int i, j;

   for (i=0;i < N;i++)
      {
        j = fprintf(stream,"%D(4,2)\n",a[i]);
        if (j < 0)
        cout <<"Number not written to file" <<a[i] <<endl;
      }
}

// read_num is passed a the array.  The values are
// read from a text file with fscanf().
// If the function is successful a 0 is returned, otherwise a
// negative value is returned (indicating an error).

void read_num(_DecimalT<4,2> b[N])
{
   int i, j;

   for (i=0;i < sizeof(b)/sizeof(b[0]);i++)
     {
```

```
      j = fscanf(stream,"%D(4,2)\n",&b[i]);
      if (j < 0)
      cout <<"Error when reading from file" <<endl;
     }

    cout <<"b[0]=" <<b[0] <<endl;
    cout <<"b[1]=" <<b[1] <<endl;
    cout <<"b[2]=" <<b[2] <<endl;
}
```

You can rewrite this program to use the `ofstream` class:

```
// This program shows how to write _DecimalT template
// constants to a file
// and scan them back again. Shows how to pass a _DecimalT
// template class array to a function.

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

#define N    3                      // Array size
                                   // for decimal declaration.

                                   // Declare valid
                                   // array.

_DecimalT<4,2> arr_1[] = {__D("12.35"), __D("25.00"),
                                   __D("-19.58")};
_DecimalT<4,2> arr_2[N];

void write_num(_DecimalT<4,2> a[N]); //Declare function to
                            // write to a file.

void read_num(_DecimalT<4,2> b[N]); //Declare function to
                             //read from a file.

int main(void)
{
   write_num(arr_1);               // Call function to write
                                 // values of the
                                 // array to outf with fprintf
                                 // library function.

   read_num(arr_2);                   // Call function to read
                                      // values of the
                                     // array from file using
                                    // fscanf() function.
}

// write_num is passed an array.  These values are
// written to a text file with the fstream class.
// If the function is successful a 0 is returned, otherwise a
// negative value is returned (indicating an error).

void write_num(_DecimalT<4,2> a[N])
{
   int i;
   ofstream outf("data",ios::trunc || ios::out,
                     filebuf::openprot);
   if (!outf)
      {
        cerr << "Could not open file 'data' " <<endl;
        exit (EXIT_FAILURE);
      }
   for (i=0; i<N; i++)
```

```
        {
          outf << a[i];
        }
        outf.close();
  }

  // read_num is passed an array.  The values are
  // read from a text file with the fstream class.
  // If the function is successful a 0 is returned, otherwise a
  // negative value is returned (indicating an error).

  void read_num(_DecimalT<4,2> b[N])
  {
     int i;
     ifstream file("data");

     if (!file)
        {
          cerr << "Could not open file 'data' " <<endl;
          exit (EXIT_FAILURE);
        }
     for (i=0; i<N; i++)
        {
          file >> b[i];
          cout << "b["<< i <<"]=" <<b[i] <<endl;
        }
     if (file.eof())
        {
          cerr << "Unexpected EOF!" <<endl;
          exit (EXIT_FAILURE);
        }

        file.close();
  }
```

The output from both programs is:

```
b[0]=12.35
b[1]=25.00
b[2]=-19.58
```

Appendix A. Using the IBM Open Class Library Source Code

This chapter provides information that helps you install and use the VisualAge for C++ for AS/400 IBM Open Class Library source code.

- "Installing the Source Code" lists the hardware requirements and explains how to install the source code.
- "Building the Source Code" on page 256 explains how to build the IBM Open Class Library source code.
- Appendix B, "Class to Implementation File Cross-Reference Table" on page 263 displays the IBM Open Class Library implementation files provided and the corresponding header file.

The source code can help you in several ways:

- You can read the source code and the developers' comments within it to gain a more in-depth understanding of how the IBM Open Class Library works. You might find this additional knowledge helpful in using these class libraries to develop your own applications.
- You can create a debug version of the IBM Open Class Library source to debug your applications that use the IBM Open Class Library.
- You can write code to extend the IBM Open Class Library and build your own versions to use in your enterprise.

IBM Open Class Library provides you with source code for the Collection, and Data Type and exception Classes.

Installing the Source Code

This section contains information that you need to install the source code.

Disk Space Required

The source code requires 30 MB for .CPP files.

To install the source code files, do the following:

1. From the workstation drive that you installed the VisualAge for C++ for AS/400 product, change the directory to

   `QIBM\PRODDATA\VISUALAG\CPPWIN\MRM\IOCSRC`

2. Copy all the files and subdirectories to your workstation into the directory CTTASW/IOCSRC. If you have VisualAge for C++ for AS/400 installed in another location, you might want to install the source code there.

The IOCSRC directory contains the:

- Collection class library source code

© Copyright IBM Corp. 1995, 1996

Building the Source Code

- Data Type and Exception class library source code
- Batch files to build the class source code

Building the Source Code

The Data Type and Exception Class Library and the Collection Class Library have dependencies on each other.

The Data Type and Exception Class Library and the Collection Class Library are compiled with the header files to produce `*MODULE` objects on the AS/400, which are then bound to produce an AS/400 service program (a `*SRVPGM` object).

Since the Data Type and Exception Class Library and the Collection Class Library have mutual dependencies, any attempt to bind either one in the absence of the other results in error messages reporting unresolved symbols. These steps are therefore required:

1. The Data Type and Exception Class Library is linked with the old Collection Class Library service program in the bind path; this interim Data Type and Exception Class Library is then provided to the Collection Class Library
2. The Collection Class Library is bound with the interim Data Type and Exception Class Library to produce the new Collection Class Library
3. The Data Type and Exception Class Library is re-bound with the new Collection Class Library to produce the new Data Type and Exception Class Library

The following section describes the procedure for building the Data Type and Exception Class Library.

Compiling Data Type and Exception Class Library Source Code

Follow these steps to compile the Data Type and Exception Class Library source code:

1. Connect to an AS/400 and set the current library:

```
cttconn /hTORAS805
ctthcmd chgcurlib <yourlib>
```

2. Invoke the VisualAge for C++ for AS/400 compiler:

```
iccas e:\as4v3r7\src\*.cpp  /O40 /I. /FT- /DIC_400 /DIC_EXPORTB=
                            /DIC_NONGUI /DIC_SAFE  /D__INO_OBJSTR__
```

Compiling Collection Class Library Source Code

Follow these steps to compile the Collection Class Library source code:

1. Connect to AS/400 and set current library

```
cttconn /hTORAS805
```

2. Invoke the VisualAge for C++ for AS/400 compiler:

```
iccas  /ASl<ccllib> /O40 /Ft- /DIC_400 /DIC_EXPORTB=
       /D__LIB_BUILD__ /I. E:\CC4V3R7\SRC\*.CPP
```

Binding

Follow these steps to bind the Data Type and Exception Class Library and the Collection Class Library:

1. Create a new Data Type and Exception Class Library service program. On the initial bind, you do not have a Collection Class Library service program available, so there are unresolved references. Also, the existing Data Type and Exception Class Library and the Collection Class Library (QSYS/Collection Class Library and QSYS/Data Type and Exception Class Library) are still in the search path, so duplicate-definition errors occur.

   Use compiler options `*UNRSLVREF`, `*DUPPROC`, `*DUPVAR` and `*NOWARN` to allow creation of an Data Type and Exception Class Library service program despite having unresolved symbols and duplicate definitions.

   The compiler option `DETAIL(*EXTENDED)` creates an output file with the results of the CRTSRVPGM command that can be viewed through the `WRKSPLF` command.

   Since this is the first bind in a two-step process, you can call the initial service program QYPPASLPRIM (short for QYPPASL PRIME):

```
// Specify that a *SRVPGM is to be created for all *MODULE
// objects in library <yourlib>
iccas b"
CRTSRVPGM SRVPGM(<yourlib>/QYPPASLPRIM) MODULE(<yourlib>/*ALL)
          EXPORT(*ALL)  OPTION(*DUPPROC *DUPVAR *UNRSLVREF *NOWARN)
          DETAIL(*EXTENDED)  TEXT('ASC prime for CCL build')"
```

2. Create a final Collection Class Library service program with resolved symbols.

```
// Specify that a *SRVPGM is to be created for all *MODULE
// objects in library <ccllib>
iccas b"CRTSRVPGM SRVPGM(<ccllib>/QYPPCC370) MODULE(<ccllib>/*ALL)
          BNDSRVPGM(<yourlib>/QYPPASLPRIM)
          EXPORT(*ALL)  OPTION(*DUPPROC *DUPVAR *UNRSLVREF *NOWARN)
          DETAIL(*EXTENDED)  TEXT('CCL final service program')"
```

3. Re-bind the Data Type and Exception Class Library. Since the new Collection Class Library is now pointed to in the binding directory and specified on the CRTSRVPGM call, this Data Type and Exception Class Library does not have unresolved symbols. The old Data Type and Exception Class Library and Collection Class Library are still in the search path, and you still receive duplicate definitions:

```
iccas b"CRTSRVPGM SRVPGM(<yourlib>/QYPPAS370) MODULE(<yourlib>/*ALL)
         EXPORT(*ALL)  BNDSRVPGM(<ccllib>/QYPPCC370)
         OPTION(*DUPPROC *DUPVAR *NOWARN)  DETAIL(*EXTENDED)
         TEXT('ASC final build using CCL final build')"
```

4. Re-bind the Collection Class Library Since the new Data Type and Exception Class Library is now pointed to in the binding directory and specified on the CRTSRVPGM call, this Collection Class Library does not have unresolved symbols. The old Data Type and Exception Class Library and Collection Class Library are still in the search path, and you still receive duplicate definitions:

```
// Specify that a *SRVPGM is to be created for all *MODULE
// objects in library <ccllib>
iccas b"CRTSRVPGM SRVPGM(<ccllib>/QYPPCC370)
MODULE(<ccllib>/*ALL)
         BNDSRVPGM(<yourlib>/QYPPASLPRIM)
         EXPORT(*ALL)  OPTION(*DUPPROC *DUPVAR *UNRSLVREF
*NOWARN)
         DETAIL(*EXTENDED)  TEXT('CCL final service program')"
```

Note: There is a system binding directory called QYPPLR370 in library QSYS that points to the standard libraries in QSYS such as the Data Type and Exception Class Library and the Collection Class Library. When you create programs that use your new Data Type and Exception Class Library and the new Collection Class Library make sure you use the create service program BNDSRVPGM parameter `BNDSRVPGM(<yourlib>/QYPPASLPRIM <yourlib>/QYPPCC370)` and the compiler options `*DUPPROC`, `*DUPVAR` and `*NOWARN` so that you do not receive duplicate definition errors.

Batch Files

You can use batch files to create the Data Type and Exception Class Library and the Collection Class Library.

The following is a sample batch file to create the Data Type and Exception Class Library. The batch file is called `ascbld.bat` and can be found in the directory `\QIBM\PRODDATA\VISUALAG\CPPWIN\MRM\IOCSRC`.

```
@rem  Sample batch file for creating the Application support class library
@rem
@rem  This sample batch file builds service program acsbld/qyppas370
@rem
@rem  if the service program cclbld/qyppcc370 does not exist yet,
@rem    then
@rem       you should make a copy of this batch file, remove the bndsrvpgm(cclbld/qyppcc370) keyword
@rem       and run it first.
@rem       run the batch file cclbld
@rem       run this file again.
@rem


@echo ibase.cpp       >  bldtmp
@echo i0string.cpp    >> bldtmp
@echo ibufferd.cpp    >> bldtmp
@echo ibuffer0.cpp    >> bldtmp
@echo ibuffer1.cpp    >> bldtmp
@echo ibuffer2.cpp    >> bldtmp
```

```
@echo ibuffer5.cpp    >> bldtmp
@echo ibuffer6.cpp    >> bldtmp
@echo ibuffer7.cpp    >> bldtmp
@echo idate.cpp       >> bldtmp
@echo idbcsb0.cpp     >> bldtmp
@echo idbcsb1.cpp     >> bldtmp
@echo idbcsb5.cpp     >> bldtmp
@echo idbcsb6.cpp     >> bldtmp
@echo idbcsb7.cpp     >> bldtmp
@echo iexcbase.cpp    >> bldtmp
@echo iexcept.cpp     >> bldtmp
@echo imsgtext.cpp    >> bldtmp
@echo imprintf.cpp    >> bldtmp
@echo istparse.cpp    >> bldtmp
@echo istpdata.cpp    >> bldtmp
@echo istring0.cpp    >> bldtmp
@echo istring1.cpp    >> bldtmp
@echo istring2.cpp    >> bldtmp
@echo istring3.cpp    >> bldtmp
@echo istring4.cpp    >> bldtmp
@echo istring5.cpp    >> bldtmp
@echo istring6.cpp    >> bldtmp
@echo istring7.cpp    >> bldtmp
@echo istring8.cpp    >> bldtmp
@echo istring9.cpp    >> bldtmp
@echo istrtest.cpp    >> bldtmp
@echo itime.cpp       >> bldtmp
@echo itrace.cpp      >> bldtmp
@echo ivbase.cpp      >> bldtmp
@echo inlsutil.cpp    >> bldtmp
@echo inotifev.cpp    >> bldtmp
@echo iobservr.cpp    >> bldtmp
@echo iobslist.cpp    >> bldtmp
@echo istdntfy.cpp    >> bldtmp
@echo itmstamp.cpp    >> bldtmp
ctthcmd crtlib ascbld
iccas -o40 -i . -ft- -dIC_400 -dIC_EXPORTB= -dIC_NONGUI -dIC_SAFE -d__INO_OBJSTR__ -aslascbld
-b"crtsrvpgm ascbld/qyppas370 export(*all)  option(*dupvar *dupproc *unrslvref *nowarn)
bndsrvpgm(cclbld/qyppcc370)" @bldtmp
```

The following is a sample batch file to create the Collection Class Library. The batch file is called `cclbld.bat` and can be found in the directory `\QIBM\PRODDATA\VISUALAG\CPPWIN\MRM\IOCSRC`.

```
@rem  Sample batch file for creating the Collection class library
@rem
@rem  This sample batch file builds service program cclbld/qyppcc370
@rem
@rem  if the service program ascbld/qyppas370 does not exist yet,
@rem    then
@rem       you should make a copy of this batch file, remove the bndsrvpgm(ascbld/qyppas370) keyword
@rem       and run it first.
@rem       run the batch file ascbld
@rem       run this file again.
@rem


@echo iiakb.cpp           >  bldtmp
@echo iiaks.cpp           >> bldtmp
@echo iiaksb.cpp          >> bldtmp
@echo iiakss.cpp          >> bldtmp
@echo iiamap.cpp          >> bldtmp
@echo iiarel.cpp          >> bldtmp
@echo iiasb.cpp           >> bldtmp
@echo iiaseq.cpp          >> bldtmp
@echo iikbdil.cpp         >> bldtmp
```

```
@echo iikbhsh.cpp          >> bldtmp
@echo iikblst.cpp          >> bldtmp
@echo iikbtab.cpp          >> bldtmp
@echo iiksavl.cpp          >> bldtmp
@echo iiksbdil.cpp         >> bldtmp
@echo iiksblst.cpp         >> bldtmp
@echo iiksbst.cpp          >> bldtmp
@echo iimwt.cpp            >> bldtmp
@echo iireldil.cpp         >> bldtmp
@echo iirelhsh.cpp         >> bldtmp
@echo iirellst.cpp         >> bldtmp
@echo iireltab.cpp         >> bldtmp
@echo iisbdil.cpp          >> bldtmp
@echo iisblst.cpp          >> bldtmp
@echo iisbtab.cpp          >> bldtmp
@echo iisrtab.cpp          >> bldtmp
@echo iissavl.cpp          >> bldtmp
@echo iissbst.cpp          >> bldtmp
@echo iissdil.cpp          >> bldtmp
@echo iisslst.cpp          >> bldtmp
@echo iisstab.cpp          >> bldtmp
@echo iivcllct.cpp         >> bldtmp
@echo imcllct.cpp          >> bldtmp
@echo ibcllct.cpp          >> bldtmp
@echo ibtree.cpp           >> bldtmp
@echo iexc.cpp             >> bldtmp
@echo iexctxt.cpp          >> bldtmp
@echo iiabag.cpp           >> bldtmp
@echo iiacllct.cpp         >> bldtmp
@echo iiaes.cpp            >> bldtmp
@echo iiahp.cpp            >> bldtmp
@echo iibagtab.cpp         >> bldtmp
@echo iiesdil.cpp          >> bldtmp
@echo iieslst.cpp          >> bldtmp
@echo iiestab.cpp          >> bldtmp
@echo iiexc.cpp            >> bldtmp
@echo iihpdil.cpp          >> bldtmp
@echo iihplst.cpp          >> bldtmp
@echo iihptab.cpp          >> bldtmp
@echo iiksstab.cpp         >> bldtmp
@echo iikstab.cpp          >> bldtmp
@echo iimapavl.cpp         >> bldtmp
@echo iimapbst.cpp         >> bldtmp
@echo iimapdil.cpp         >> bldtmp
@echo iimaphsh.cpp         >> bldtmp
@echo iimaplst.cpp         >> bldtmp
@echo iimaptab.cpp         >> bldtmp
@echo iisettab.cpp         >> bldtmp
@echo iismavl.cpp          >> bldtmp
@echo iismbst.cpp          >> bldtmp
@echo iismdil.cpp          >> bldtmp
@echo iismlst.cpp          >> bldtmp
@echo iismtab.cpp          >> bldtmp
@echo iisrdil.cpp          >> bldtmp
@echo iisrlst.cpp          >> bldtmp
@echo iiaset.cpp           >> bldtmp
@echo iiasm.cpp            >> bldtmp
@echo iiasr.cpp            >> bldtmp
@echo iiass.cpp            >> bldtmp
@echo iiatree.cpp          >> bldtmp
@echo iibagdil.cpp         >> bldtmp
@echo iibaghsh.cpp         >> bldtmp
@echo iibaglst.cpp         >> bldtmp
@echo iiksbtab.cpp         >> bldtmp
@echo iiksdil.cpp          >> bldtmp
@echo iikshsh.cpp          >> bldtmp
@echo iikslst.cpp          >> bldtmp
```

```
@echo iikssavl.cpp         >> bldtmp
@echo iikssbst.cpp         >> bldtmp
@echo iikssdil.cpp         >> bldtmp
@echo iiksslst.cpp         >> bldtmp
@echo iiseqdil.cpp         >> bldtmp
@echo iiseqlst.cpp         >> bldtmp
@echo iiseqtab.cpp         >> bldtmp
@echo iisetavl.cpp         >> bldtmp
@echo iisetbst.cpp         >> bldtmp
@echo iisetdil.cpp         >> bldtmp
@echo iisethsh.cpp         >> bldtmp
@echo iisetlst.cpp         >> bldtmp
@echo imtree.cpp           >> bldtmp
ctthcmd crtlib cclbld
iccas -o40 -i . -ft- -aslcclbld -dIC_400 -dIC_EXPORTB=
-d__LIB_BUILD__ -b"crtsrvpgm cclbld/qyppcc370
export(*all) option(*dupvar *dupproc *nowarn *unrslvref)
bndsrvpgm(ascbld/qyppas370)"  @bldtmp
```

Building the Source Code

Appendix B. Class to Implementation File Cross-Reference Table

The following table shows the implementation file that corresponds to each class. An asterisk in the .cpp file name shows that there are multiple files for that class.

Class Name	Implementation File
I0String	i0string.cpp
IAccessError	iexcbase.cpp
IApplicator	iiter.h
IAssertionFailure	iexcbase.cpp
IAutoElemPointer	iptr.h
IAutoPointer	iptr.h
IABag	iabag.h
IACollection	iacllct.h
IADeque	iadqu.h
IAEqualityCollection	iaequal.h
IAEqualityKeyCollection	iaeqkey.h
IAEqualityKeySortedCollection	iaeqksrt.h
IAEqualitySequence	iaes.h
IAEqualitySortedCollection	iaeqsrt.h
IAHeap	iahp.h
IAKeyBag	iakb.h
IAKeyCollection	iakey.h
IAKeySet	iaks.h
IAKeySortedBag	iaksb.h
IAKeySortedCollection	iaksrt.h
IAKeySortedSet	iakss.h
IAMap	iamap.h
IAMultiwayTree	iamwt.h
IAOrderedCollection	iaorder.h
IAPriorityQueue	iapqu.h
IAQueue	iaqu.h
IARelation	iarel.h
IASequence	iaseq.h
IASequentialCollection	iasqntl.h
IASet	iaset.h
IASortedBag	iasb.h

© Copyright IBM Corp. 1995, 1996

Class Implementation File

Class Name	Implementation File
IASortedCollection	iasrt.h
IASortedMap	iasm.h
IASortedRelation	iasr.h
IASortedSet	iass.h
IAStack	iastk.h
IBag	ibag.h
IBagAsAVLTree	ibagavl.h
IBagAsBstTree	ibagbst.h
IBagAsDilTable	ibagdil.h
IBagAsHshTable	ibaghsh.h
IBagAsList	ibaglst.h
IBagAsTable	ibagtab.h
IConstantApplicator	iiter.h
ICursor	icursor.h
IDeque	idqu.h
IDequeAsDilTable	idqudil.h
IDequeAsList	idqulst.h
IDequeAsTable	idqutab.h
IElementCursor	icursor.h
IElemPointer	iptr.h
IEqualitySequence	ies.h
IEqualitySequenceAsDilTable	iesdil.h
IEqualitySequenceAsList	ieslst.h
IEqualitySequenceAsTable	iestab.h
IGBag	ibag.h
IGBagAsAVLTree	ibagavl.h
IGBagAsBstTree	ibagbst.h
IGBagAsDilTable	ibagdil.h
IGBagAsHshTable	ibaghsh.h
IGBagAsList	ibaglst.h
IGBagAsTable	ibagtab.h
IBase	ibase.cpp
IBase::Version	ibase.cpp
IBaseErrorInfo	iexcept.cpp
IDBCSBuffer	idbcsb*.cpp
IDate	idate.cpp

Class Name	Implementation File
IDeviceError	iexcbase.cpp
IEvent	ievent.cpp
IEventData	ievent.inl
IEventParameter1	ievent.inl
IEventParameter2	ievent.inl
IEventResult	ievent.inl
IException	iexcbase.cpp
IException::TraceFn	iexcbase.cpp
IExceptionLocation	iexcbase.cpp
IGDeque	idqu.h
IGDequeAsDilTable	idqudil.h
IGDequeAsList	idqulst.h
IGDequeAsTable	idqutab.h
IGDilTable	iseqdil.h
IGEqualitySequence	ies.h
IGEqualitySequenceAsDilTable	iesdil.h
IGEqualitySequenceAsList	ieslst.h
IGEqualitySequenceAsTable	iestab.h
IGHeap	ihp.h
IGHeapAsDilTable	ihpdil.h
IGHeapAsList	ihplst.h
IGHeapAsTable	ihptab.h
IGKeyBag	ikb.h
IGKeyBagAsHshTable	ikbhsh.h
IGKeySet	iks.h
IGKeySetAsAvlTree	iksavl.h
IGKeySetAsBstTree	iksbst.h
IGKeySetAsDilTable	iksdil.h
IGKeySetAsHshTable	ikshsh.h
IGKeySetAsList	ikslst.h
IGKeySetAsTable	ikstab.h
IGKeySortedBag	iksb.h
IGKeySortedBagAsDilTable	iksbdil.h
IGKeySortedBagAsList	iksblst.h
IGKeySortedBagAsTable	iksbtab.h
IGKeySortedSet	ikss.h

Class Implementation File

Class Name	Implementation File
IGKeySortedSetAsAvlTree	ikssavl.h
IGKeySortedSetAsBstTree	ikssbst.h
IGKeySortedSetAsDilTable	ikssdil.h
IGKeySortedSetAsList	iksslst.h
IGKeySortedSetAsTable	iksstab.h
IGMap	imap.h
IGMapAsAvlTree	imapavl.h
IGMapAsBstTree	imapbst.h
IGMapAsDilTable	imapdil.h
IGMapAsHshTable	imaphsh.h
IGMapAsList	imaplst.h
IGMapAsTable	imaptab.h
IGMultiwayTree	imwt.h
IGPriorityQueue	ipqu.h
IGPriorityQueueAsDilTable	ipqudil.h
IGPriorityQueueAsList	ipqulst.h
IGPriorityQueueAsTable	ipqutab.h
IGQueue	iqu.h
IGQueueAsDilTable	iqudil.h
IGQueueAsList	iqulst.h
IGQueueAsTable	iqutab.h
IGRelation	irel.h
IGSequence	iseq.h
IGSequenceAsList	iseqlst.h
IGSequenceAsTable	iseqtab.h
IGSet	iset.h
IGSetAsAvlTree	isetavl.h
IGSetAsBstTree	isetbst.h
IGSetAsDilTable	isetdil.h
IGSetAsHshTable	isethsh.h
IGSetAsList	isetlst.h
IGSetAsTable	isettab.h
IGSortedBag	isb.h
IGSortedBagAsAvlTree	isbavl.h
IGSortedBagAsBstTree	isbbst.h
IGSortedBagAsDilTable	isbdil.h

Class Implementation File

Class Name	Implementation File
IGSortedBagAsList	isblst.h
IGSortedBagAsTable	isbtab.h
IGSortedMap	ism.h
IGSortedMapAsAvlTree	ismavl.h
IGSortedMapAsBstTree	ismbst.h
IGSortedMapAsDilTable	ismdil.h
IGSortedMapAsList	ismlst.h
IGSortedMapAsTable	ismtab.h
IGSortedRelation	isr.h
IGSortedRelationAsDilTable	isrdil.h
IGSortedRelationAsList	isrslst.h
IGSortedRelationAsTable	isrtab.h
IGSortedSet	iss.h
IGSortedSetAsAvlTree	issavl.h
IGSortedSetAsBstTree	issbst.h
IGSortedSetAsDilTable	issdil.h
IGSortedSetAsList	isslst.h
IGSortedSetAsTable	isstab.h
IGStack	istk.h
IGStackAsDilTable	istkdil.h
IGStackAsList	istklst.h
IGStackAsTable	istktab.h
IHeap	ihp.h
IHeapAsDilTable	ihpdil.h
IHeapAsList	ihplst.h
IHeapAsTable	ihptab.h
IKeyBag	ikb.h
IKeyBagAsHshTable	ikbhsh.h
IKeySet	iks.h
IKeySetAsAvlTree	iksavl.h
IKeySetAsBstTree	iksbst.h
IKeySetAsDilTable	iksdil.h
IKeySetAsHshTable	ikshsh.h
IKeySetAsList	ikslst.h
IKeySetAsTable	ikstab.h
IKeySortedBag	iksb.h

Class Implementation File

Class Name	Implementation File
IKeySortedBagAsDilTable	iksbdil.h
IKeySortedBagAsList	iksblst.h
IKeySortedBagAsTable	iksbtab.h
IKeySortedSet	ikss.h
IKeySortedSetAsAvlTree	ikssavl.h
IKeySortedSetAsBstTree	ikssbst.h
IKeySortedSetAsDilTable	ikssdil.h
IKeySortedSetAsList	iksslst.h
IKeySortedSetAsTable	ikssstab.h
IMap	imap.h
IMapAsAvlTree	imapavl.h
IMapAsBstTree	imapbst.h
IMapAsDilTable	imapdil.h
IMapAsHshTable	imaphsh.h
IMapAsList	imaplst.h
IMapAsTable	imaptab.h
IMessageText	imsgtext.cpp
IMngElemPointer	iptr.h
IMngPointer	iptr.h
IMultiwayTree	imwt.h
INotificationEvent	inotifev.cpp
INotifier	istdntfy.cpp
IObserver	iobservr.cpp
IObserverList	iobslist.cpp
IObserverList::Cursor	iobslist.cpp
IOrderedCursor	icursor.h
IOutOfMemory	iexcbase.cpp
IOutOfSystemResource	iexcbase.cpp
IPriorityQueue	ipqu.h
IPriorityQueueAsDilTable	ipqudil.h
IPriorityQueueAsList	ipqulst.h
IPriorityQueueAsTable	ipqutab.h
IQueue	iqu.h
IQueueAsDilTable	iqudil.h
IQueueAsList	iqulst.h
IQueueAsTable	iqutab.h

Class Implementation File

Class Name	Implementation File
IRefCounted	irefcnt.cpp
IReference	irefcnt.cpp
IRelation	irel.h
ISequence	iseq.h
ISequenceAsDilTable	iseqdil.h
ISequenceAsList	iseqlst.h
ISequenceAsTable	iseqtab.h
ISet	iset.h
ISetAsAvlTree	isetavl.h
ISetAsBstTree	isetbst.h
ISetAsDilTable	isetdil.h
ISetAsHshTable	isethsh.h
ISetAsList	isetlst.h
ISetAsTable	isettab.h
ISortedBag	isb.h
ISortedBagAsAvlTree	isbavl.h
ISortedBagAsBstTree	isbbst.h
ISortedBagAsDilTable	isbdil.h
ISortedBagAsList	isblst.h
ISortedBagAsTable	isbtab.h
ISortedMap	ism.h
ISortedMapAsAvlTree	ismavl.h
ISortedMapAsBstTree	ismbst.h
ISortedMapAsDilTable	ismdil.h
ISortedMapAsList	ismlst.h
ISortedMapAsTable	ismtab.h
ISortedRelation	isr.h
ISortedRelationAsDilTable	isrdil.h
ISortedRelationAsList	isrslst.h
ISortedRelationAsTable	isrtab.h
ISortedSet	iss.h
ISortedSetAsAvlTree	issavl.h
ISortedSetAsBstTree	issbst.h
ISortedSetAsDilTable	issdil.h
ISortedSetAsList	isslst.h
ISortedSetAsTable	isstab.h

Class Implementation File

Class Name	Implementation File
IStack	istk.h
IStackAsDilTable	istkdil.h
IStackAsList	istklst.h
IStackAsTable	istktab.h
IStandardNotifier	istdntfy.cpp
IString	istring*.cpp
IStringEnum	istrenum.cpp
IStringParser	istparse.cpp
IStringParser::SkipWords	istparse.cpp
IStringTest	istrtest.cpp
IStringTestMemberFn	istrtest.cpp
ISystemErrorInfo	iexcept.cpp
ITime	itime.cpp
ITimer	itimer.cpp
ITimer::Cursor	itimer.cpp
ITimerFn	itimer.cpp
ITimerMemberFn	itimer.cpp
ITimerMemberFn0	itimer.cpp
ITrace	itrace.cpp
ITreeCursor	ibtree.h
IVBase	ivbase.cpp
IXLibErrorInfo	iexcept.cpp

Part 6. Glossary, Bibliography and Index

© Copyright IBM Corp. 1995, 1996

Glossary

This glossary defines terms and abbreviations that are used in this book. If you do not find the term you are looking for, refer to the *IBM Dictionary of Computing*, SC20-1699.

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.

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. 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.

American National Standard Code for Information Interchange (ASCII). See *ASCII*

American National Standards Institute (ANSI). See *ANSI*

ANSI (American National Standards Institute). An organization consisting of producers, consumers, and general interest groups, that establishes the procedures by which accredited organizations create and maintain voluntary industry standards in the United States.

applicator class. A class that provides a function that may be applied to the elements of a collection during iteration with the `allElementsDo()` function.

array. In C++, an aggregate that consists of data objects, with identical attributes, each of which may be uniquely referenced by subscripting.

array implementation. Implementation of an abstract data type using an array. Also called a tabular implementation.

ASCII (American National Standard Code for Information Interchange). The standard code, using a coded character set consisting of 7-bit coded characters (8 bits including parity check), that is used for information interchange among data processing systems, data communication systems, and associated equipment. The ASCII set consists of control characters and graphic characters.

Note: IBM has defined an extension to ASCII code (characters 128-255).

automatic storage. Storage that is allocated on entry to a routine or block and is freed on the subsequent return. Sometimes referred to as *stack storage* or *dynamic storage*.

automatic storage management. The process that automatically allocates and deallocates objects in order to use memory efficiently.

auxiliary classes. Classes that support other classes. Auxilliary classes in the Collection Class Library include classes for cursors, pointers and applicators.

AVL tree. A balanced binary search tree that does not allow the height of two siblings to differ by more than one.

B

B* tree (B star tree). A tree in which only the leaves contain whole elements. All other nodes contain keys.

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.

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. (2) The unit of data transmitted to and from a

© Copyright IBM Corp. 1995, 1996

device. Each block contains one record, part of a record, or several records.

bounded collection. A collection that has an upper limit on the number of elements it can contain.

built-in. A function that the compiler will automatically inline instead of making the function call.

C

C++ class library. A collection of C++ classes.

character array. An array of type **char**.

child. A node that is subordinate to another node in a tree structure. Only the root node is not a child.

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 their members. (2) A user-defined data type. A class data type can contain both data representations (data members) and functions (member functions).

class library. A collection of C++ classes.

class template. A blueprint describing how a set of related classes can be constructed.

collection. (1) In a general sense, an implementation of an abstract data type for storing elements. (2) An abstract class without any ordering, element properties, or key properties. All abstract classes are derived from collection.

Complex Mathematics library. A C++ class library that provides the facilities to manipulate complex numbers and perform standard mathematical operations on them.

concrete class. A class that implements an abstract data type but does not allow polymorphism.

const. (1) An attribute of a data object that declares the object cannot be changed. (2) A keyword that allows you to define a variable whose value does not change.

constructor. A special C++ class member function that has the same name as the class and is used to construct and possibly initialize class objects.

containment function. A function that determines whether a collection contains a given element.

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 due to conversion since accuracy of data representation varies among different data types. (2) The process of changing from one form of representation to another; for example to change from decimal representation to binary representation. (3) A change in the type of a value. For example, when you add values having different data types, the compiler converts both values to a common form before adding the values.

conversion function. A member function that specifies a conversion from its class type to another type.

copy constructor. A constructor that copies a class object of the same class type.

cursor. A reference to an element at a specific position in a data structure.

cursor iteration. The process of repeatedly moving the cursor to the next element in a collection until some condition is satisfied.

D

data member. The smallest possible piece of complete data. Elements are composed of data members.

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.

declaration. (1) A description that makes an external object or function available for use without defining it.

default class. A class with preprogrammed definitions that can be used for simple implementations.

default constructor. A constructor that takes no arguments, or one that, if it takes arguments, all its arguments have default values.

default implementation. One of several possible implementation variants offered as the default for a specific abstract data type.

default operation class. A class with preprogrammed definitions for all required element and key operations for a particular implementation.

degree. The number of children of a node.

deque. A queue that can have elements added and removed at both ends. A double-ended queue.

dequeue. An operation that removes the first element of a queue.

derivation. The relationship between a class and the classes above or below it in a class hierarchy.

derived class. A class that inherits from one or more base classes. 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.

destructor. A special member function that has the same name as its class, preceded by a tilde (˜), and that "cleans up" after an object of that class, for example by freeing storage that was allocated when the object was created. A destructor has no arguments and no return type.

difference. Given two sets A and B, the difference (A-B) is the set of all elements contained in A but not in B.

diluted array. An array in which elements are deleted by being flagged as deleted, rather than by actually removing them from the array and shifting later elements to the left.

diluted sequence. A sequence implemented using a diluted array.

double-byte character set (DBCS). See *DBCS*.

dynamic. Pertaining to an operation that occurs at the time it is needed rather than at a predetermined or fixed time such as program load time.

dynamic allocation. Assignment of system resources to a program when the program is executed rather than when it is loaded into main storage.

E

EBCDIC (extended binary-coded decimal interchange code). A coded character set of 256 8-bit characters.

element. The component of an array, subrange, enumeration, or set.

element equality. A relation that determines whether two elements are equal.

element function. A function, called by a member function, that accesses the elements of a class.

enqueue. An operation that adds an element as the last element to a queue.

equality collection. (1) An abstract class with the property of element equality. (2) In general, any collection that has element equality.

equality key collection. An abstract class with the properties of element equality and key equality.

equality key sorted collection. An abstract class with the properties of element equality, key equality, and sorted elements.

equality sequence. A sequentially ordered flat collection with element equality.

equality sorted collection. An abstract class with the properties of element equality and sorted elements.

exception. (1) Any user, logic, or system error that a function detects but does not itself deal with. Typically the function passes the error on to a handling routine (also called throwing the exception).

extended binary-coded data interchange code (EBCDIC). A coded character set of 256 8-bit characters.

F

file descriptor. A small positive integer that the system uses instead of the file name to identify an open file.

file scope. A name declared outside all blocks and classes has file scope and can be used after the point of declaration in a source file.

filter. A command whose operation consists of reading data from standard input or a list of input files and

writing data to standard output. Typically, its function is to perform some transformation on the data stream.

first element. The element visited first in an iteration over a collection. Each collection has its own definition for first element. For example, the first element of a sorted set is the element with the smallest value.

flat collection. A collection that has no hierarchical structure.

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`.

G

global. Pertaining to information available to more than one program or subroutine.

H

hash function. A function that determines which category, or bucket, to put an element in. A hash function is needed when implementing a hash table.

hash table. A data structure that divides all elements into (preferably) equal-sized categories, or buckets, to allow quick access to the elements. The hash function determines which bucket an element belongs in.

header file. A file that contains declarations for classes, functions, or data that are defined elsewhere. Also known as an *include* file.

heap. An unordered flat collection that allows duplicate elements.

height of a tree. The length of the longest path from the root to a leaf.

I

I/O Stream library. A class library that provides the facilities to deal with many varieties of input and output.

implementation class. A class that implements a concrete class. Implementation classes are never used directly.

indirection. A mechanism for connecting objects by storing, in one object, a reference to another object.

initializer. An expression used to initialize data objects. In C++, there are three types of initializers:

1. An expression followed by an assignment operator is used to initialize fundamental data type objects or class objects that have copy constructors.
2. An expression enclosed in braces ({ }) is used to initialize aggregates.
3. A parenthesized expression list is used to initialize base classes and members using constructors.

input stream. A stream used to read input.

instance. 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.

intersection. Given collections A and B, the set of elements that is contained in both A and B.

iteration. The process of repeatedly applying a function to a series of elements in a collection until some condition is satisfied.

iteration order. The order in which elements are accessed when iterating over a collection. In ordered collections, the element at position 1 will be accessed first, then the element at position 2, and so on. In sorted collections, the elements are accessed according to the ordering relation provided for the element type. In collections that are not ordered the elements are accessed in an arbitrary order. Each element is accessed exactly once.

K

key access. A property that allows elements to be accessed by matching keys.

key bag. An unordered flat collection that uses keys and can contain duplicate elements.

key collection. (1) An abstract class that has the property of key access. (2) In general, any collection that uses keys.

key equality. A relation that determines whether two keys are equal.

key() function. When used on a flat collection, a function that returns a reference to the key of an element.

key-type function. Any of several functions of an element type, that are used by the Collection Class Library member functions to manipulate the keys of a class.

key set. An unordered flat collection that uses keys and does not allow duplicate elements.

key sorted bag. A sorted flat collection that uses keys and allows duplicate elements.

key sorted collection. An abstract class with the properties of key equality and sorted elements.

key sorted set. A sorted flat collection that uses keys and does not allow duplicate elements.

L

last element. The element accessed last in an iteration over a collection. Each collection has its own definition for last element. For example, the last element of a sorted set is the element with the largest value.

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. In a tree, nodes without children. Synonymous with terminals.

linked implementation. An implementation in which each element contains a reference to the next element in the collection. Pointer chains are used to access elements in linked implementations. Linked implementations are also called linked list implementations.

linked sequence. A sequence that uses a linked implementation.

linker. (1) A computer program for creating load modules from one or more object modules or load modules by resolving cross references among the modules and, if necessary, adjusting addresses. (2) Synonym for linkage editor.

local. (1) In C++, pertaining to the relationship between a language object and a block such that the language object has a scope contained in that block.

locale. The definition of the subset of a user's environment that depends on language and cultural conventions.

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.

manipulator. A value that can be inserted into streams or extracted from streams to affect or query the behavior of the stream.

mask. A pattern of bits or characters that controls the keeping, deleting, or testing of portions of another pattern of bits or characters.

MBCS. See *multibyte character set*

member. A data object or function in a structure, union, or class. Members can also be classes, enumerations, bit fields, and type names.

member function. An operator or function that is declared as a member of a class. A member function has access to the private and protected data members and member functions of objects of its class. Member functions are also called methods.

memory leak. A potentially error-causing situation that occurs when dynamic memory that has been allocated to an application is not freed by the application when the memory is no longer needed.

method. In C++, a synonym for member function.

mode. A collection of attributes that specifies a file's type and its access permissions.

multibyte character set (MBCS). A character set whose characters consist of more than 1 byte. Used in languages such as Japanese, Chinese, and Korean, where the 256 possible values of a single-byte character set are not sufficient to represent all possible characters.

multiple inheritance. An object-oriented programming technique implemented in C++ 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.

N

n-ary tree. A tree that has an upper limit, *n*, imposed on the number of children allowed for a node.

nested class. A class defined within the scope of another class.

node. In a tree structure, a point at which subordinate items of data originate.

NULL. A pointer that does not point to a data object.

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, and to represent the end of a C++ character string.

O

object. (1) An instance of a class. (2) In object-oriented design or programming, an abstraction consisting of data and the operations associated with that data. See also *class*. (3) A region of storage. An object is created when a variable is defined or new is invoked. An object is destroyed when it goes out of scope. (See also *instance*.)

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.

operand. An entity on which an operation is performed.

operation class. A class that defines all required element and key operations required by a specific collection implementation.

operator function. An overloaded operator that is either a member of a class or that takes at least one argument that is a class type or a reference to a class type.

ordered collection. (1) An abstract class that has the property of ordered elements. (2) In general, any collection that has its elements arranged so that there is always a first element, last element, next element, and previous element.

ordering relation. A property that determines how the elements are sorted. Ascending order is an example of an ordering relation.

overflow. A condition that occurs when a portion of the result of an operation exceeds the capacity of the intended unit of storage.

overloading. An object-oriented programming technique that allows you to redefine functions and most standard C++ operators when the functions and operators are used with class types.

P

parent node. A node to which one or more other nodes are subordinate.

pointer class. A class that implements pointers.

polymorphism. The technique of taking an abstract view of an object or function and using any concrete objects or arguments that are derived from this abstract view.

positioning property. The property of an element that is used to position the element in a collection. For example, the value of the key may be used as the positioning property.

precedence. The priority system for grouping different types of operators with their operands.

precondition. A condition that a function requires to be true when it is called.

predicate function. A function that returns an IBoolean value of *true* or *false*. (IBoolean is an integer-represented boolean type.)

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.

priority queue. A queue that has a priority assigned to its elements. When accessing elements, the element with the highest priority is removed first. A priority queue has a largest-in, first-out behavior.

private. Pertaining to a class member that is 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.

profiling. The process of generating a statistical analysis of a program that shows processor time and the percentage of program execution time used by each procedure in the program.

property function. A function that is used to determine whether the element it is applied to has a given property or characteristic. A property function can be used, for example, to remove all elements with a given property.

protected. Pertaining to a class member that is only accessible to member functions and friends of that class, or to member functions and friends of classes derived from that class.

prototype. A function declaration or definition that includes both the return type of the function and the types of its parameters. See *function prototype*.

public. Pertaining to a class member that is accessible to all functions.

Q

queue. A sequence with restricted access in which elements can only be added at the back end (or bottom) and removed from the front end (or top). A queue is characterized by first-in, first-out behavior and chronological order.

R

reference class. A class that links a concrete class to an abstract class. Reference classes make polymorphism possible with the Collection Classes.

relation. An unordered flat collection class that uses keys, allows for duplicate elements, and has element equality.

returned element. An element returned by a function as the return value.

root. A node that has no parent. All other nodes of a tree are descendants of the root.

S

SBCS. See *single-byte character set*.

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.

sequence. A sequentially ordered flat collection.

sequential collection. An abstract class with the property of sequentially ordered elements.

siblings. All the children of a node are said to be siblings of one another.

signal. (1) A condition that may or may not be reported during program execution. (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.

single-byte character set (SBCS). A set of characters in which each character is represented by a 1-byte code.

sorted bag. A sorted flat collection that allows duplicate elements.

sorted collection. (1) An abstract class with the property of sorted elements. (2) In general, any collection with sorted elements.

sorted map. A sorted flat collection with key and element equality.

sorted relation. A sorted flat collection that uses keys, has element equality, and allows duplicate elements.

sorted set. A sorted flat collection with element equality.

stack. A data structure in which new elements are added to and removed from the top of the structure. A stack is characterized by Last-In-First-Out (LIFO) behavior.

standard error. An output stream usually intended to be used for diagnostic messages.

standard input. An input stream usually intended to be used for primary data input. Standard input comes from the keyboard unless redirection or piping is used, in which case standard input can be from a file or the output from another command.

standard output. An output stream usually intended to be used for primary data output.

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.

stream. (1) A continuous stream of data elements being transmitted, or intended for transmission, in character or binary-digit form, using a defined format. (2) A file access object that allows access to an ordered sequence of characters, as described by the ISO C standard. A stream provides the additional services of user-selectable buffering and formatted input and output.

stream buffer. A stream buffer is a buffer between the ultimate consumer, ultimate producer, and the I/O Stream Library functions that format data. It is implemented in the I/O Stream Library by the `streambuf` class and the classes derived from `streambuf`.

string. A contiguous sequence of characters, usually terminated by and including the first null byte.

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.

subscript. One or more expressions, each enclosed in brackets, that follow an array name. A subscript refers to an element in an array.

subtree. A tree structure created by arbitrarily denoting a node to be the root node in a tree. A subtree is always part of a whole tree.

superset. Given two sets A and B, A is a superset of B if and only if all elements of B are also elements of A. That is, A is a superset of B if B is a subset of A.

T

tabular implementation. An implementation that stores the location of elements in tables. Elements in a tabular implementation are accessed by using indices to arrays.

tabular sequence. A sequence that uses a tabular implementation.

template. A family of classes or functions with variable types.

terminals. Synonym for *leaves*.

this collection. The collection to which a function is applied.

tree. A hierarchical collection of nodes that can have an arbitrary number of references to other nodes. A unique path connects every two nodes.

try block. A block in which a known C++ exception can be passed to a handler.

type specifier. Used to indicate the data type of an object or function being declared.

typed implementation class. A class that implements a concrete class and provides an interface that is specific to a given element type. This interface allows the compiler to verify that, for example, integers cannot be added to a set of strings.

typeless implementation class. A class that implements a concrete class and provides an interface that is not specific to a given element type.

U

ultimate consumer. The target of data in an I/O operation. An ultimate consumer can be a file, a device, or an array of bytes in memory.

ultimate producer. The source of data in an I/O operation. An ultimate producer can be a file, a device, or an array of byes in memory.

unbounded collection. A collection that has no upper limit on the number of elements it can contain.

undefined cursor. A cursor that may or may not be valid, and that may or may not refer to a different element of the collection from the element it referred to before the function call that resulted in its becoming

undefined. An undefined cursor may refer to no element of the collection, and still be a valid cursor.

underflow. (1) A condition that occurs when the result of an operation is less than the smallest possible nonzero number. (2) Synonym for arithmetic underflow, monadic operation.

union. (1) A variable that can hold any one of several data types, but only one data type at a time. (2) Given the sets A and B, all elements of A, B, or both A and B.

unique collection. A collection in which the value of an element only occurs once; that is, there are no duplicate elements.

unordered collection. A collection that has no order to its elements.

unrecoverable error. An error for which recovery is impossible without use of recovery techniques external to the executing program.

V

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.

W

white space. Space characters, tab characters, form-feed characters, and new-line characters.

Bibliography

This bibliography lists the publications that make up the IBM VisualAge for C++ for AS/400 library and publications of related IBM products referenced in this book. The list of related publications is not exhaustive but should be adequate for most VisualAge for C++ for AS/400 users.

The IBM VisualAge for C++ for AS/400 Library

The following books are part of the IBM VisualAge for C++ for AS/400 library.

- *VisualAge for C++ for AS/400 Licensed Programming Specification*, GC09-2414
- *VisualAge for C++ for AS/400 Installation Guide and Product Overview*, SC09-2415
- *VisualAge for C++ for AS/400 C++ User's Guide*, SC09-2416
- *VisualAge for C++ for AS/400 IBM Open Class Library Reference*, SC09-2440
- *VisualAge for C++ for AS/400 C Library Reference*, SC09-2441
- *VisualAge for C++ for AS/400 C++ Programming Guide*, SC09-2417
- *ILE C/C++ MI Library Reference*, SC09-2418
- *VisualAge for C++ for AS/400 C++ Language Reference*, SC09-2442
- *VisualAge for C++ for AS/400 IBM Open Class Library User's Guide*, SC09-2443
- *IBM Access Class Library for OS/400 Reference*, SC41-4620
- *IBM Access Class Library for Windows Reference*, SC41-4622
- *IBM Access Class Library User's Guide*, SC41-4623
- *ILE Concepts*, SC41-4606

C and C++ Related Publications

- *Portability Guide for IBM C*, SC09-1405
- *American National Standard for Information Systems / International Standards Organization — Programming Language C (ANSI/ISO 9899-1990[1992])*
- *Draft Proposed American National Standard for Information Systems — Programming Language C++ (X3J16/92-0060)*

Other Books You Might Need

The following list contains the titles of IBM books that you might find helpful. These books are not part of the VisualAge for C++ for AS/400 or operating system libraries.

Non-IBM Publications

Many books have been written about the C++ language and related programming topics. The authors use varying approaches and emphasis. The following is a sample of some non-IBM C++ publications that are generally available. This sample is not an exhaustive list. IBM does not specifically recommend any of these books, and other C++ books may be available in your locality.

- *The Annotated C++ Reference Manual* by Margaret A. Ellis and Bjarne Stroustrup, Addison-Wesley Publishing Company.
- *C++ Primer* by Stanley B. Lippman, Addison-Wesley Publishing Company.
- *Object-Oriented Design with Applications* by Grady Booch, Benjamin/Cummings.
- *Object-Oriented Programming Using SOM and DSOM* by Christina Lau, Van Nostrand Reinhold.

© Copyright IBM Corp. 1995, 1996

Index

Special Characters

__D macro
  constants 238
_DecimalT Class Template
  constants 238
  debug macros 247
  input and output 239
  introduction 237
  mathematical operators 239
  overflow behavior 245
  runtime errors 245, 247
    _DecErr class 247
_DecimalT class template representation 237
_DecimalT template class
  calling another program 250
  constructing objects 238
  digitsof 244
  passing a pointer 249
  passing to a function 249
  precisionof 244
  sizeof 244
  writing constants to a file 251

A

absolute value of complex numbers 9
abstract Collection Classes
  based-on concept 128
  class hierarchy 87
  cursor class used with 96
  key collection
    restriction on replacing elements 95
  naming convention 88
  polymorphism 139
  relationship to other classes 86
accessing elements 81, 96
add() Collection Class function
  behavior of 93
  example of behavior 93
  properties of 92
  role of 92
addAsFirst() Collection Class function 93
addAsLast() Collection Class function 93
addAsNext() Collection Class function 93
addAsPrevious() Collection Class function 93
adding elements to collections
  effect on cursors 96
  overview 92—93
addition of complex numbers 9
addOrReplaceElementWithKey() Collection Class function 92
allElementsDo() Collection Class function 99, 100
anonymous streams 30
applicator 67
arithmetic operators for _DecimalT template class 239
assignment (Collection Class Library)
  using member functions 106
  using operation classes 110
  using separate functions 107
AT&T C++ Language System Release 1.2
  history of class libraries 1
auxiliary class 78, 84

B

bag
  description 73
  implementation variant explained 128
  properties of 79
IBase class 194
base() streambuf function 32
based-on concept in Collection Class Library
  overview 127—128
bcd.h header file 237
Binary Coded Decimal Library 237
binary conversion in IString class 206
bounded collections 102
building source code 256
  building 256

C

case change of IString objects 209
cerr predefined stream 29
IChildAlreadyExistsException 145
children of a tree node 83
cin predefined stream 29, 35
class
  categories of Collection Classes 84—89
  general types of Collection Classes 77
  hierarchy
    abstract collections 87

class (continued)
  hierarchy (continued)
    Binary Coded Decimal Library for OS/400 2
    Collection Class Library 88
    Complex Mathematics 2
    data type classes 191
    exception classes 191
    I/O Stream Classes 27
    illustrations 2
clog predefined stream 29
collection
  copying 102
  cursor association 95
  iterating over 98—102
  modifying 92—95
  referencing 102
  using polymorphism with 139
Collection Class Library
  categories of classes 84—89
  implementation strategy 84
  reasons for using 73
  steps for using 91
  structure of classes 86
  types of collections 77
compare() function
  Collection Class Library
    using operation classes 110
    using separate functions 107
comparison
  of IDate objects 226
  of ITime objects 228
  of ITimeStamp objects 230
complex class
  conversion functions 17
  mathematical functions 15
  mathematical operators 12
  review of complex numbers 9
  trigonometric functions 16
Complex Mathematics Library 9, 10
complex.h header file 10
concatenation of IString objects 202
concrete classes
  cursors with 96
  relationship to other classes 86
concrete implementations 87, 128
conjugates of complex numbers 9
constant iterator class 99—102
constants defined in bcd.h 238
constructors
  Collection Class Library
    errors 184
constructors (continued)
  Collection Class Library (continued)
    restriction on defining 106
  IDate class 225
  IString class 199
  IStringTest class 211
  ITime class 227
  ITimeStamp class 229
containment function 82
conversion
  _DecimalT template class to a _DecimalT template class 242
  _DecimalT template class to a floating point type 243
  _DecimalT template class to an integer type 243
  explicit 241
  implicit 241
  rules 241
conversion functions
  complex class 17
  IString class 199, 206
copying
  collections 102
  IString class 200
cout predefined stream 29, 38
ICursorInvalidException 145
cursors
  accessing elements with 96
  association with a collection 95
  classes 96
  classes, abstract 96
  description 95—98
  effect of replacing elements 95
  iteration 98—99
  locating elements with 96
  properties that may cause an exception 145
  reasons for using 96
  removing elements with 94
  unexpected results 179
  validity 96, 146
customizing an implementation 127—138

D

Data Type classes
  introduction 191
  I0String class 197—210
  IBase class 191
  IBaseErrorInfo class 191
  IBuffer class 191

Data Type classes *(continued)*
IDate class 225—230
IDBCSBuffer class 191
IString class 197—210
IStringTest class 210—211
ITime class 225—230
ITimeStamp class 225—230
IVBase class 194
IDate class 225—230
DBCS 195
decimal conversion in IString class 206
default classes in Collection Class Library
instantiation 91
introduction 87
naming convention 88
relationship to other classes 86
strategy for using 84
tutorial on using 176
default constructors 179
deleting substrings of IString objects 204
deque 77, 82
destructors
Collection Class Library
restriction on defining 106
diluted implementation 131, 132
diluted table 131
reasons for using 131
dispatchNotificationEvent, overview 232
double-byte character set 195, 198

E

eback() streambuf function 32
ebuf() streambuf function 32
egptr() streambuf function 32
element equality 78, 80—81
elementAt() function
accessing elements with 96
replacing elements using 97
role in Collection Class Library 95
versions of 97
elements in Collection Class Library
accessing 81, 96
adding 92—93
effect on cursors 96
functions
errors 180
introduction 105
methods for providing 106
relationship to derived classes 113
using element operation classes 109
elements in Collection Class Library *(continued)*
functions *(continued)*
using member functions 106
using pointers with 115
using separate functions 107
iterating 98—102
locating 92, 96
See also locate... functions
occurrence 94
operation classes 109
polymorphism 139
removing 93—94
effect on cursors 96
replacing 95
using elementAt() function 97
value 94
elementWithKey() Collection Class function 97
IEmptyException 146
endl manipulator 40
epptr() streambuf function 32
equality operators for _DecimalT template class 240
equality relation 80
equality sequence 76, 79
equality test
in Collection Class Library
using member functions 106
using operation classes 110
using separate functions 107
in complex class 13
error
determination in Collection Class Library 179
handling
by math.h for complex class 19
stream input 53
exception
IAccessError exception 213
IAssertionFailure exception 213
IChildAlreadyExistsException 145
ICursorInvalidException 145
IDeviceError exception 213
IEmptyException 146
IException class 147
IFullException 103, 146
hierarchy 147
IIdenticalCollectionException 146
in Collection Class Library 143—148
IInvalidParameter exception 213
IInvalidReplacementException 146
IInvalidRequest exception 213
IKeyAlreadyExistsException 146

exception (continued)
INotBoundedException 103, 146
INotContainsKeyException 146
IOutOfMemory exception 213
IOutOfSystemResource exception 213
IOutOfWindowResource exception 213
IPositionInvalidException 146
IPreconditionViolation 147
IResourceExhausted exception 147, 213
IRootAlreadyExistsException 147
tracing 179, 183
violated precondition in Collection Class Library 143
IException class 147, 179, 213—220
trace function 183
Exception Classes
IException class 213—220
ITrace class 220—223
extraction operator
See operator >>

F

file input 43
file output 45
filebuf class
header files 28
moving through a file 48
filters in I/O Stream Classes 47
firstElement() Collection Class function 97
flat collection classes
overview 77—82
with restricted access 82
flush manipulator 40
forICursor Collection Class Macro 99
format state
mutually exclusive flags 58
formatting
of IString objects 209
of output streams 55
fstream class
assigning to cin and cout 47
file input 43
file output 45
header files 28
IFullException 103, 146

G

get pointer 32
getline() istream function 37
gptr() streambuf function 32
Gregorian calendar 225

H

handleNotificationsFor, overview 231
hashing
description 137
restriction on replacing elements 95
restrictions on defining 107
using operation classes 110
using separate functions 107
header files
I/O Stream Classes 28
heap
description 76
properties of 79
replacing elements 95
hexadecimal conversion in IString class 206
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 29
stream buffers 31
IIdenticalCollectionException 146
ifstream class
file input 43
header files 28
imaginary part of a complex number 9
implementation in Collection Class Library
basic 127
concrete 87
instantiating the default 91
provided by Collection Class Library 128
replacing the default 127
tailoring 127—138
implementation variant
choosing 128
features of 130
provided by Collection Class Library 128
indexing of strings 198

inequality operator for complex class 14
inheritance in Collection Class Library 139
INotificationEvent class overview 232
INotifier class overview 232
input
  correcting errors 53
  from files 43
  from standard input 35
  IString class 201
  white space in 37
inserting substrings into IString objects 204
insertion operator
  See operator <<
installing source code 255
  installing 255
IInvalidReplacementException 146
iomanip.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
iostream_withassign class
  example of using 47
  header files 28
iostream.h header file
  classes defined 28
is... methods of IString class 207
isFull() Collection Class function 103
istream class
  header files 28
  input operator
    for class types 49
    multiple types in an input statement 36
    pointers to char 36
    white space 37
istream_withassign class
  header files 28
IString... classes
  See entries for IString... under S
istrstream class
  header files 28
isValid() function
  limitation 96
  role of 146
iteration
  over collections 98—102
  restrictions 98
  using exceptions 144
iterator class 99—102

J

Julian date format 225

K

key access
  basic properties of flat collections 78
  description 80
  errors 179, 181, 182
  overview 80—82
  restriction on defining 106
  using operation classes 110
  using separate functions 107
key bag 74, 79, 93
key collection 95
  restriction on replacing elements 95
key equality 80—81
key set
  adding elements 93
  description 75
  properties of 79
key sorted bag 74, 79
key sorted set 75, 79
key-type functions
  errors 179, 180, 182
  global 182
  introduction 105
  methods for providing 106
  relationship to derived classes 113
  using element operation classes 109
  using member functions 106
  using pointers with 115
  using separate functions 107
IKeyAlreadyExistsException 146

L

linked implementation 96
list 131
  description 131
locateOrAddElementWithKey() Collection Class
  function 92
locating elements 96
lowercase and IString objects 209

M

- macros for the exception classes 217
- magnitude of complex numbers 9
- managed pointer class 78, 84
  - relationship to derived classes 113
- map 75
- mathematical functions for complex class 15
- matherr() library function 19
- maxNumberOfElements() Collection Class function 103
- memory management
  - restriction on defining 107
  - using operation classes 110
  - using separate functions 107
- modifying a collection 92—95
- multiple collections 78, 82
- multiple inheritance in Collection Class Library 139
- multiplication of complex numbers 9
- mutually exclusive format flags 58

N

- n-ary tree class 83
- naming conventions 88
- National Language Support (NLS) 195
- newCursor() function
  - abstract classes 96
- NLS 195
- node of a tree 83
- INotBoundedException 103, 146
- INotContainsKeyException 146
- notification class hierarchy 234
- notification ID overview 232
- notification protocol description 233
- notifier protocol overview 231
- notifyObservers, overview 232
- null character 202
- numeric conversion in IString class 206

O

- object definition, default implementation 91
- observer protocol overview 231
- obsolete functions 1
- ofstream class
  - file output 45
  - header files 28
- operations classes
  - using 109—113
- operator +
  - complex class 9
  - IString class 202
- operator <
  - Collection Class Library 106
- operator <<
  - defining for class types 51
  - in I/O Stream Classes 25
  - ostream class 66
  - IString class 201
  - ITime class 228
- operator =
  - Collection Class Library 106
- operator ==
  - Collection Class Library 106
  - complex class 13
- operator >>
  - defining for class types 49
  - in I/O Stream Classes 25
  - istream class 66
  - multiple types in an input statement 36
  - pointers to char 36
  - IString class 201
- ordered collection
  - multiple inheritance 139
  - removing an element 94
- ordering relation
  - as a collection property 78
  - possible orderings of collections 79
  - restriction on replacing elements 95
  - sorted collections 79
  - using member functions 106
- ostream class
  - header files 28
  - output operator
    - for class types 51
    - 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 209
- parameterized manipulators
  - and simple manipulators 65

parameterized manipulators *(continued)*
example 68
for your own types 67
introduction 65
parent in a tree 83
pbase() streambuf function 32
pointer class
managed
See managed pointer class
relationship to derived classes 113
role of 78
using with element and key-type functions 115
polar representation of complex numbers 9
polymorphism 84, 139
using pointers to elements 115
positioning property 95
IPositionInvalidException 146
pptr() streambuf function 32
precondition
violated 143—145
IPreconditionViolation exception 147
predefined streams
assigning fstream objects to 47
defined by iostream.h 29, 35, 38
predicate functions in Collection Class Library 94
priority queue 77, 82
problem determination
Collection Class Library 179
put pointer 32
putback 32

Q

queue 77
queue collection 82

R

real part of a complex number 9
reference class
naming convention 88
relationship to other classes 86
referencing a collection 102
relation 76
sorted relation 76
relational operators for _DecimalT template class 240
remove() function
Collection Class Library
behavior of 94
role of 92
removeAll() Collection Class function 94
removeFirst() Collection Class function 94
removeLast() Collection Class function 94
removing elements
See also remove... functions for collections
effect on cursors 96
overview 93—94
replace() Collection Class function 92, 95
replaceAt() function
role of 95
replacing
substrings of IString objects 204
replacing elements 95
See also replace... functions for collections
using elementAt() function 97
IRootAlreadyExistsException 147

S

separate functions in Collection Class Library 107—109
sequence 76
sequence as list
template with element operation class 109
sequential collections
replacing elements 95
set 74
simple manipulators 65
sorted bag 74, 79
sorted collections
multiple inheritance 139
ordering relation 79
sorted map
description 75
properties of 79
restrictions for adding elements 92
sorted relation
properties of 79
sorted set 75, 79
source code 255
stack 77, 82
standard error 38
standard input 35
standard output 38
stderr 38
stdin 35
stdiobuf class
header files 28
stdiostr.h 28
stdiostream class
header files 28

stdout 38
stopHandlingNotificationsFor, overview 232
stream buffers
  definition 31
  implementation 32
  purpose 31
Stream Library 1
stream.h header file 28
streambuf class
  header files 28
  member functions 32
string buffers 198
IString class
  binary conversion 206
  concatenating objects of 202
  decimal conversion 206
  deleting a substring 204
  formatting 209
  hexadecimal conversion 206
  indexing of strings 198
  inserting a substring 204
  is... methods of IString class 207
  numeric conversion 206
  padding 209
  replacing a substring 204
  string length 205
  testing characteristics of IString objects 207
  using 197—210
  word count 205
string input 36
IStringTest class 210—211
IStringTestMemberFn class 211
stripping blanks from IString objects 209
strstrea.h 28
strstream class
  header files 28
  possible uses of 63
strstreambuf class
  header files 28
substrings
  creating from IString objects 200
  deleting in IString objects 204
  finding within IString objects 202
subtraction of complex numbers 9
system failures 144
system restrictions 144

T

table 131
table implementation 131
table sequence 131
tabular implementation 96
tailoring an implementation 127—138
templates
  arguments
    declaration errors 179, 183, 184
  linking with 185
  operation class inheritance 110
ITime class 225—230
ITimeStamp class 225—230
ITrace class 213—223
trace macros 221
tree 78, 83
trigonometric functions for complex class 16
tutorials 149—177
typed implementation class
  naming convention 88
  purpose 87
typeless implementation class
  purpose 88
  relationship to other classes 86

U

ultimate consumer 31
ultimate producer 31
unbounded collections 102
unique collections
  adding elements 93
  compared to multiple collections 78
  description 82
Unix System Laboratories C++ Language System 1
unordered collections
  characteristics 79
  cursor iteration drawbacks 99
uppercase and IString objects 209
user-defined input operator 35, 49
user-defined output operator 39, 51

V

variant classes
  See also implementation variant
  description 87
  naming convention 88
  strategy for using 84

variant classes *(continued)*
    tailoring a collection with 127—138
IVBase class 194
void* type 94

W

white space
    in IString objects 209
    in string input 37
word index 198
words
    finding within IString objects 202