VisualAge C++ Professional for AIX

Programming Tasks and Library Reference

Version 6.0

SC09-4963-00

Note!

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

Edition Notice (June, 2002)

This edition applies to Version 6.0 of IBM VisualAge C++ Professional for AIX (5675–F56) 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.

© Copyright International Business Machines Corporation 1998, 2002. All rights reserved.
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Notices vii
Programming Interface Information ix
Trademarks and Service Marks ix
Industry Standards x

About This Book. xi

Chapter 1. Program Stream I/O 1
Stream Processing 1
Standard Streams 1
Redirect Standard Streams 2
File Handles for Standard Streams 3
I/O Buffering 3
Considerations for Programming Stream I/O . . . 4

Chapter 2. Data Mapping and Storage . . 7
Format of Double-Byte Character Data. 7
Format of Eight-Byte Floating Point Data 7
Format of Four-Byte Integer Data 8
Format of Single-Byte Character Data 8
Data Mapping 8
Mapping of Fundamental Data Types 9
Mapping of Compound Data Types. 9
MacIntosh and Twobyte Alignment Rules 10
Alignment Rules for Nested Aggregates 12
Packed Alignment Rules 13
RISC System/6000 Alignment Rules 14
Storage of float and double Types 15
Storage of int, long, and short Types 16

Chapter 3. Signals and Exception Handling 19
Choose Signal Handlers 19
Signal Handling 19
Signals 20
Program Signal Handling. 20
Signal Handling Considerations 21
Example of Using Volatile Variables 22

Chapter 4. Using Memory Heaps. . . . 25
Memory Management Functions 25
Managing Memory with Multiple Heaps 27
Types of Memory 29
Debugging Memory Heaps 29
Create and Use a Fixed Size Heap 31
Create and Use an Expandable Heap 33
Debug Programs with Heap Memory. 36
Change the Default Heap Used in a Program . . . 37
Example of Creating and Using a User Heap . . . 38
Example of Creating and Using a Shared-Memory User Heap. 39

Chapter 5. Program Optimization . . . 45
Overview of Optimization 45
Optimization Techniques Used by VisualAge C++ 46
Enhanced Handling of Math and String Library Functions 48
Find Faster I/O Techniques 48
Optimize Your Application 49
Reduce Function-Call Overhead 50
Coding Techniques That Can Improve Performance 51
Memory Management and Performance 53
Mixed-Mode Arithmetic 53
Expressions 53
Variables and Optimization 54
Optimize String Manipulation 55

Chapter 6. Floating Point Operations 57
Floating Point Hardware 57
Compile-Time Floating-Point Arithmetic. 58
Rounding Mode Restrictions. 59

Chapter 7. USL Input/Output Stream Classes 61
USL I/O Streaming. 61
The USL I/O Stream Class Hierarchy. 62
USL I/O Stream Header Files 63
Open a File for Input and Read from the File . . . 66
Open a File for Output and Write to the File . . . 67
Manipulate Strings with the strstream Classes . . . 68
Stream Buffers 70
Format State Flags 71
Manipulators 71
Thread Safety and USL I/O Streaming 72
Create Manipulators 74
  Define an APP Parameterized Manipulator . . . 74
  Define a MANIP Parameterized Manipulator . . 75
  Define Nonassociative Parameterized Manipulators 76

Chapter 8. USL Complex Math Classes 77
Complex Mathematics Library Overview 77
Review of Complex Numbers 77
Header Files and Constants for the complex and c_exception Classes. 78
Mathematical Operators for complex 79
Friend Functions for complex 80
Input and Output Operators for complex 81
Error Functions 81
Construct complex Objects 82
  Use complex Input and Output Operators . . . 82
  Use Friend Functions with complex 84
Handle complex Mathematics Errors. 86
Example: Calculate Roots. 88
Example: Use Equality and Inequality Operators . . 90

Chapter 9. Other Utilities 93
c++filt Name Demangling Utility 93

© Copyright IBM Corp. 1998, 2002

CreateExportList Command 96
linkxlC Command 96
Constructing a Library 96
makeC++SharedLib Command 100
Initialize Shared Library 102
Specify Priority Levels for Library Objects . . 103
Example of Object Initialization in a Group of Files 104
loadAndInit Routine 105
Format 105
Description 105
Parameters 105
Return Values 106
terminateAndUnload Routine 106
Format 106
Description 106
Parameters 107
Return Values 107
Error Codes 107
_makepath — Create Path 107
_splitpath — Decompose Path Name 109

Appendix. Non-ISO USL Classes . . . 111

complex 111
complex - Hierarchy List 111
complex - Member Functions and Data by Group 111
complex - Associated Globals 113
complex - Inherited Member Functions and Data 118
c_exception 118
c_exception - Hierarchy List 118
c_exception - Member Functions and Data by Group 119
c_exception - Associated Globals 120
c_exception - Inherited Member Functions and Data 122
filebuf 122
filebuf - Hierarchy List 122
filebuf - Member Functions and Data by Group 122
filebuf - Inherited Member Functions and Data 128
fstream 128
fstream - Hierarchy List 129
fstream - Member Functions and Data by Group 129
fstream - Inherited Member Functions and Data 132
fstreambase 133
fstreambase - Hierarchy List 133
fstreambase - Member Functions and Data by Group 133
fstreambase - Inherited Member Functions and Data 137
ifstream 138
ifstream - Hierarchy List. 138
ifstream - Member Functions and Data by Group 138
ifstream - Inherited Member Functions and Data 142
ios 143
ios - Hierarchy List 143
ios - Member Functions and Data by Group . . 143
ios - Enumerations 151
ios - Inherited Member Functions and Data . . 152
iostream 153
iostream - Hierarchy List 153
iostream - Member Functions and Data by Group 153
iostream - Inherited Member Functions and Data 157
iostream_withassign 158
iostream_withassign - Hierarchy List 158
iostream_withassign - Member Functions and Data by Group 158
iostream_withassign - Inherited Member Functions and Data 163
istream 164
istream - Hierarchy List 164
istream - Member Functions and Data by Group 165
istream - Inherited Member Functions and Data 188
istream_withassign 188
istream_withassign - Hierarchy List 188
istream_withassign - Member Functions and Data by Group 188
istream_withassign - Inherited Member Functions and Data 193
istrstream 194
istrstream - Hierarchy List 194
istrstream - Member Functions and Data by Group 194
istrstream - Inherited Member Functions and Data 199
ofstream 200
ofstream - Hierarchy List 200
ofstream - Member Functions and Data by Group 200
ofstream - Inherited Member Functions and Data 204
ostream 205
ostream - Hierarchy List. 205
ostream - Member Functions and Data by Group 205
ostream - Inherited Member Functions and Data 218
ostream_withassign 218
ostream_withassign - Hierarchy List. 218
ostream_withassign - Member Functions and Data by Group 219
ostream_withassign - Inherited Member Functions and Data 222
ostrstream 223
ostrstream - Hierarchy List 223
ostrstream - Member Functions and Data by Group 223
ostrstream - Inherited Member Functions and Data 228
stdiobuf 228
stdiobuf - Hierarchy List 228
stdiobuf - Member Functions and Data by Group 228
stdiobuf - Inherited Member Functions and Data 231
stdiostream 231
stdiostream - Hierarchy List 231
stdiostream - Member Functions and Data by Group 232
stdiostream - Inherited Member Functions and Data 234

streambuf 234
streambuf - Hierarchy List 235
streambuf - Member Functions and Data by Group 235
streambuf - Inherited Member Functions and Data 247
strstream 248
strstream - Hierarchy List 248
strstream - Member Functions and Data by Group 248
strstream - Inherited Member Functions and Data 252
strstreambase 253
strstreambase - Hierarchy List 253
strstreambase - Member Functions and Data by Group 253
strstreambase - Inherited Member Functions and Data 255
strstreambuf 256
strstreambuf - Hierarchy List 256
strstreambuf - Member Functions and Data by Group 256
strstreambuf - Inherited Member Functions and Data 263

Appendix. MEMDBG Library Functions 265

_debug_calloc — Allocate and Initialize Memory 265
_debug_free — Free Allocated Memory. 266
_debug_heapmin — Free Unused Memory in the Default Heap 268
_debug_malloc — Allocate Memory 269
_debug_memcpy — Copy Bytes 271
_debug_memmove — Copy Bytes 273
_debug_memset — Set Bytes to Value 274
_debug_realloc — Reallocate Memory Block . . . 276
_debug_strcat — Concatenate Strings 278
_debug_strcpy — Copy Strings 279
_debug_strnset — Set Characters in String. . . . 281
_debug_strncat — Concatenate Strings 282
_debug_strncpy — Copy Strings 284
_debug_strset — Set Characters in String 285
_debug_ucalloc — Reserve and Initialize Memory from User Heap 287
_debug_uheapmin — Free Unused Memory in User Heap 289
_debug_umalloc — Reserve Memory Blocks from User Heap 290

Contacting IBM 293

Comments on This Help. 293
Fee Support 293
Consulting Services 294

Notices

Note to U.S. Government Users Restricted Rights -- use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described 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:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

© Copyright IBM Corp. 1998, 2002

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

Lab Director
IBM Canada Ltd.
8200 Warden Avenue
Markham, Ontario L6G 1C7
Canada

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains 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 of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. 1998, 2002. All rights reserved.

Programming Interface Information

Programming interface information is intended to help you create application software using this program.

General-use programming interface allow the customer to write application software that obtain the services of this program's tools.

However, this information may also contain diagnosis, modification, and tuning information. Diagnosis, modification, and tuning information is provided to help you debug your application software.

Warning: Do not use this diagnosis, modification, and tuning information as a programming interface because it is subject to change.

Trademarks and Service Marks

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

AIX
AS/400
DB2
CICS
C Set ++
IBM
iSeries
Network Station
Object Connection
OS/2
OS/390
OS/400
Open Class
Operating System/2
Operating System/400
PowerPC 403
PowerPC 601
PowerPC 603
PowerPC 604
Presentation Manager
RS/6000
S/390
SAA
Systems Application Architechture
TeamConnection
VisualAge
WebSphere
Workplace Shell

Lotus, Lotus Notes, and Domino are trademarks or registered trademarks of the Lotus Development Corporation in the United States, or other countries, or both.

Tivoli Management Environment, TME 10, and Tivoli Module Designer are trademarks of Tivoli Systems Inc. in the United States, or other countries, or both.

Encina and DCE Encina Lightweight Client are trademarks of Transarc Corporation in the United States, or other countries, or both.

Microsoft, Win32, Windows, Windows NT, and the Windows logo are trademarks or registered trademarks of Microsoft Corporation in the United States, or other countries, or both.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

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

ActionMedia, LANDesk, MMX, Pentium and ProShare are trademarks or registered trademarks of Intel Corporation in the United States, or other countries, or both.

C-bus is a registered trademark of Corollary, Inc.

PC Direct is a registered tradmark of Ziff Communicatoins Company and is used by IBM Corporation under license

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

Industry Standards

VisualAge C++ Professional for AIX, Version 6.0 supports the following standards:

- The C language is consistent with the International Standard for Information Systems-Programming Language C (ANSI/ISO-IEC 9899–1999(E)).
- The IBM Systems Application Architecture (SAA) C Level 2 language definition.
- The C++ language is consistent with the International Standard for Information Systems-Programming Language C++ (ISO/IEC 14882:1998).
- The ISO/IEC 9945–1:1990/IEEE POSIX 1003.-1990 standard.
- The X/Open Common Applications Environment Specifications, System Interfaces and Headers, Issue 4.

About This Book

The information in this PDF document is also available in the online help.

To find this information, or any topics listed in this document as Related Concepts, Related Tasks, or Related References, simply type the topic title into the search bar in the top frame of your browser in the online help.

For some topics, the suggested references may already be contained in this document. In such cases, there is a cross-reference to the page on which the related topic appears.

© Copyright IBM Corp. 1998, 2002

Chapter 1. Program Stream I/O

Stream Processing

Input and output are mapped into logical data streams, either text or binary. Streams present a consistent view of file contents, independent of the underlying file system. VisualAge C++provides I/O buffering to increase the efficiency of system level I/O.

Text Streams
Text streams contain printable characters and control characters.

There may not be an exact correspondence between the characters in a stream and the output. The VisualAge C++compiler may add, alter, or ignore some new-line characters during input or output so that they conform to the conventions for representing text in the operating system environment. Printable characters are not changed.

On output, each new-line character is translated into a carriage-return character, followed by a line-feed character. On input, a line-feed character or a carriage-return character followed by a line-feed character is converted to a new-line character.

Binary Streams A binary stream is a sequence of characters or data. The data is not altered on input or output.

The Ctrl-Z character is treated like any other character and does not indicate end-of-file.

RELATED CONCEPTS
Standard Streams (page 1)

RELATED TASKS
Redirect Standard Streams (page 2)

RELATED REFERENCES
File Handles for Standard Streams (page 3)
I/O Buffering (page 3)

Standard Streams

VisualAge C++supports the C standard streams and C++ iostreams.

C Standard Streams
Any program that includes the header stdio.h can use the C standard streams for I/O. The following streams are automatically set up by the run-time environment:

- stdin
  The input device from which your application normally retrieves its data. For example, the library function getchar uses stdin.
- stdout
  The output device to which your application normally directs its output. For example, the library function printf uses stdout.

© Copyright IBM Corp. 1998, 2002

- stderr
  The output device to which your application directs its diagnostic messages.

C++ iostreams

C++ VisualAge C++provides 2 versions of the C++ iostreams:

- Streams as implemented in previous versions of VisualAge C++. These streams will be used by C++ source files which #include any of the following header files:
  <fstream.h>, <iomanip.h>, <stdiostr.h>, <stream.h>, and <strstream.h>.
  The functions declared in these files are not reentrant.
- The input/output library component of the ISO Standard C++ Library. These streams will be used by C++ source files which #include any of the following header files:
  <fstream>, <iomanip>, <ios>, <iosfwd>, <iostream>, <istream>, <ostream>, <sstream>, <streambuf>, and <strstream>.

C++ The input streams are istream and wistream objects. The output streams have type ostream and wostream. The names of the wide character streams and classes start with a "w". The iostream standard stream objects are:

- cin and wcin
  The standard narrow- and wide- character input streams.
- cout and wcout
  The standard narrow- and wide- character output streams.
- cerr and wcerr
  The standard error streams. Output to these streams is unit-buffered. Characters sent to these streams are flushed after each insertion operation.
- **clog and wclog**
  Additional standard error streams. Output to these streams is fully buffered.

RELATED CONCEPTS

Stream Processing (page 1)

RELATED TASKS

Redirect Standard Streams (page 2)

RELATED REFERENCES

I/O Buffering (page 3)
File Handles for Standard Streams (page 3)

Redirect Standard Streams

By default, the standard streams read from the keyboard and write to the screen. When you launch a program from the operating system GUI, or want input and output operations on these streams to read from and write to files, you can redirect the standard streams.

There are two ways to do this:

- **From Within an Application**
  To redirect C standard streams to a file from within your application, use the

freopen library function. For example, to send your output to a file called pia.out instead of sending it to stdout, code the following statement in your program:

```
freopen("pia.out", "w", stdout);
```

- **From the Command Line**
  To redirect C or C++ standard streams to a file from the command line, use the standard redirection symbols > and < with the file handles for standard streams. For example, to run the program bill.exe, which has two required parameters *XYZ* and *123*, and redirect the output from stdout to a file called bill.out, use the following command:

```
bill XYZ 123 > bill.out
```

You can also use the file handles to redirect one standard stream to another. For example, to redirect stderr to stdout, use the command:

```
2 > &1
```

RELATED CONCEPTS

Stream Processing (page 1)
Standard Streams (page 1)

RELATED REFERENCES

File Handles for Standard Streams (page 3)

File Handles for Standard Streams

The operating system associates a file handle with each of the streams as follows:

File Handle	C Stream	C++ Stream
0	stdin	cin and wcin
1	stdout	cout and wcout
2	stderr	cerr, clog, wcerr, and wclog

The file handle and stream are not equivalent. There may be situations where a file handle is associated with a different stream. For example, file handle 2 may be associated with a stream other than stderr, cerr, or clog.

RELATED CONCEPTS

Stream Processing (page 1)
Standard Streams (page 1)

RELATED TASKS

Considerations for Programming Stream I/O (page 4)
Redirect Standard Streams (page 2)

I/O Buffering

VisualAge C++ buffers stream I/O to increase the efficiency of system-level I/O. The following buffering modes are used:

Unbuffered
: Characters are transmitted as soon as possible. This mode is also called unit buffered.

Line buffered
: Characters are transmitted as a block when a new-line character is encountered or when the buffer is filled.

Fully buffered
: Characters are transmitted as a block when the buffer is filled.

The buffering mode specifies the manner in which the buffer is flushed, if a buffer exists. Streams are fully buffered by default unless they are connected to a character device such as the keyboard or an operating system pipe. Streams for character devices are line buffered.

C++ Standard iostreams cerr, and wcerr are unbuffered, but clog and wclog are fully buffered.

Your programs should take advantage of buffering to increase the efficiency of system-level I/O. To ensure your output appears in the expected order and is complete, you may have to control the buffering explicitly. Programs that use C stream I/O can control buffering in the following ways:

- Call the fflush function to clear the buffer. If the last operation on the stream is a read operation, fflush discards the unread portion of the buffer. If the last operation on the stream is a write operation, fflush writes out the contents of the buffer.
- Call the fclose function to flush the buffer for a file and then close it.

If your program terminates normally, VisualAge C++ automatically closes all files and flushes all buffers. When a program ends abnormally, all files are closed but the buffers are not flushed.

You can change the buffering mode of a stream from within your code. You must do this before performing any operation on the file.

To ensure data is transmitted to external storage as soon as possible, use the setvbuf or setbuf function to set the buffering mode to unbuffered. Call these functions after the file is open and before performing read or write operations on the file. You can also use setvbuf and setbuf to control buffering in other ways; setvbuf is the more flexible of the two functions.

The default buffer size is 4096 bytes. To specify a different initial size for the buffer allocated for the stream, set the blksize parameter of the fopen function when you open the stream.

Disk caching performed by the operating system can also affect the time when characters are actually transferred to and from a physical disk.

RELATED CONCEPTS
Stream Processing (page 1)

Considerations for Programming Stream I/O

Using Standard Streams
Standard streams are supported by the C library and the C++ iostream library.

To use the C standard streams, include the header <stdio.h> or, in C++, the header <cstdio>.

The standard streams are not available when you are using the subsystem libraries.

On input and output operations requiring a file pointer, you can use the standard streams in the same manner as you would any other file pointer.

The standard streams are always in text mode at the start of your program. You can change the mode to binary or back to text, without redirecting the stream, by calling the freopen function with no file name specified.

Limitations
VisualAge C++does not support record level I/O.

You cannot call the ftell , fseek, fgetpos, fsetpos, and rewind functions to get or change the file position within character devices or operating system pipes.

You cannot seek past the end of a text file. Seeking past the end of a binary file that was opened in mode w, w+, wb+, w+b, or wb, creates a new end-of-file position and writes nulls between the old end-of-file position and the new one.

RELATED CONCEPTS
Stream Processing (page 1)
Standard Streams (page 1)

RELATED TASKS
Redirect Standard Streams (page 2)

RELATED REFERENCES
I/O Buffering (page 3)
File Handles for Standard Streams (page 3)

Chapter 2. Data Mapping and Storage

Format of Double-Byte Character Data

Type	wchar_t
Alignment	2-byte aligned
Storage Mapping	Stored in 2 bytes

RELATED CONCEPTS

Data Mapping (page 8)

RELATED REFERENCES

Mapping of Fundamental Data Types (page 9)
Format of Single-Byte Character Data (page 8)

Format of Eight-Byte Floating Point Data

VisualAge C++ conforms to IEEE format, in which a floating point number is represented in terms of sign (S), exponent (E), and fraction (F):

$$(-1)^S \times 2^E \times 1.F$$

<table>
<tr><td>Type</td><td>double</td></tr>
<tr><td>Alignment</td><td>WIN 8-byte aligned</td></tr>
<tr><td>Bit Mapping</td><td>In the internal representation, there is 1 bit for the sign (S), 11 bits for the exponent (E), and 52 bits for the fraction (F). The bits are mapped with the fraction in bit 0 to bit 51, the exponent in bit 52 to bit 62, and the sign in bit 63:

6 66655555555
3 21098765432
S EEEEEEEEEEE

5544444444443333333333222222222211111111110000000000
1098765432109876543210987654321098765432109876543210
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF

Read the lines vertically from top to bottom. For example, the third column of numbers shows that bit 61 is part of the exponent.</td></tr>
<tr><td>Storage Mapping</td><td>In the following mapping, high memory is to the right.
<table>
<tr><th>byte 0</th><th>byte 1</th><th>byte 2</th><th>...</th><th>byte 5</th><th>byte 6</th><th>byte 7</th></tr>
<tr><td>
76543210</td><td>111111
54321098</td><td>22221111
32109876</td><td>...</td><td>44444444
76543210</td><td>55555544
54321098</td><td>66665555
32109876</td></tr>
<tr><td>FFFFFFFF</td><td>FFFFFFFF</td><td>FFFFFFFF</td><td>...</td><td>FFFFFFFF</td><td>EEEEFFFF</td><td>SEEEEEEE</td></tr>
</table></td></tr>
</table>

RELATED CONCEPTS

Data Mapping (page 8)

© Copyright IBM Corp. 1998, 2002

RELATED REFERENCES

Mapping of Fundamental Data Types (page 9)
Mapping of Compound Data Types (page 9)

Format of Four-Byte Integer Data

Type	long, int, and their signed and unsigned counterparts
Alignment	4-byte aligned
Storage Mapping	Byte-reversed, for example, 0x4A5D3B2C (where 2C is the least significant byte and 4A is the most significant byte) is represented in storage as shown below. High memory is to the right. byte 0: 2C, byte 1: 3B, byte 2: 5D, byte 3: 4A

RELATED CONCEPTS

Data Mapping (page 8)

Format of Single-Byte Character Data

Type	signed char, unsigned char, bool
Alignment	Byte-aligned
Storage Mapping	Stored in 1 byte

RELATED CONCEPTS

Data Mapping (page 8)

Data Mapping

Each data format supported by VisualAge C++is mapped into storage with a specific alignment.

Alignment refers to the positioning of variables on byte boundaries. Alignment within a structure can be changed with #pragma align, #pragma pack, the __align() specifier, the __attribute__((aligned)) specifier or with the -qalign compiler option.

For automatic variables, consider the following information:

- Automatic variables have the same mapping as other variables.
- When optimization is turned on, automatic variables are ordered to minimize padding.
- Automatic variables are always mapped on the stack instead of a data segment. Because memory on the stack is constantly reallocated on the stack, automatic variables are *not*guaranteed to be retained after the return of the function that used them.

RELATED REFERENCES

Mapping of Fundamental Data Types (page 9)
Mapping of Compound Data Types (page 9)
Macintosh and Twobyte Alignment Rules (page 10)
Alignment Rules for Nested Aggregates (page 12)
Packed Alignment Rules (page 12)
RISC System-6000 Alignment Rules (page 14)

Mapping of Fundamental Data Types

The following table lists the data mapping formats used for C and C++ fundamental data types.

Type	Mapping
bool char signed char unsigned char	Single-Byte Character
wchar_t	Double-Byte Character (32 bits on AIX 5.1 or later in 64-bit mode)
short int unsigned short int	Two-Byte Integer
int unsigned int long int unsigned long int	Four-Byte Integer (32-bit mode) (long is 8-byte in 64-bit mode)
long long	Eight-Byte Integer
float	Four-Byte Floating Point
double	Eight-Byte Floating Point
long double	Eight-Byte Floating Point if compiled with the -qlongdouble option. Sixteen-Byte Floating Point if compiled with -qnolongdouble.

RELATED CONCEPTS

Data Mapping (page 8)

RELATED REFERENCES

Mapping of Compound Data Types (page 9)
-qlongdouble Compiler Option

Mapping of Compound Data Types

The list below includes compound data types for which you can access the allocated storage in C and C++ programs.

C++ The C++ compiler may generate extra fields for classes that contain base classes or virtual functions. Objects of these types may not conform to the mappings listed below for structures.

- Null-Terminated Character Strings
- Fixed-Length Arrays Containing Simple Data Types
- Aligned Structures
- Unaligned or Packed Structures

- Arrays of Structures

RELATED CONCEPTS

Data Mapping (page 8)

RELATED REFERENCES

Mapping of Fundamental Data Types (page 9)

MacIntosh and Twobyte Alignment Rules

All unions and structures are halfword aligned regardless of their members. Within the aggregate, members are aligned according to their type. The size of types for the Macintosh system is the same as on the RISC System/6000 system. The table below summarizes alignment information for each type.

Type, Size, and Alignment for the Macintosh System		
Type	**Alignment**	**Size**
char	byte aligned	byte
short	halfword aligned	halfword
(long) int	halfword aligned	word in 32-bit mode doubleword in 64-bit mode
long long int	halfword aligned	doubleword
pointer	halfword aligned	word
float	halfword aligned	word
double	halfword aligned	doubleword
long double	halfword aligned	doubleword
long double with the -qlongdouble compiler option.	halfword aligned	quadrupleword

Example

The following example uses these symbols to show padding and boundaries:

```
p = padding
| = halfword boundary
: = byte boundary
```

For:

```
#pragma options align=mac68k
struct A {
  char a;
}

sizeof(A) == 2
```

The layout of A is:

```
|a:p|
```

For:

```
#pragma options align=mac68k
struct B {
  char a;
```

```
  double b;
}

sizeof(B) == 10
```

The layout of B is:

```
|a:p|b:b|b:b|b:b|b:b|
```

Bit fields for Macintosh Format

The following rules apply when you are laying out bit fields in structures.

- An individual bit field can be at most 32 bits long.
- Bit fields are packed into a word and are aligned on a 2-byte boundary.
- Bit fields that would cross a word boundary are moved to the *next* halfword boundary even if they are already starting on a halfword boundary. (The bit field may still end up crossing a word boundary.)
- A bit field of width zero forces the next member (even if it is not a bit field) to start at the *next* halfword boundary even if the zero-width bit field is currently at a halfword boundary.
- A structure containing nothing but zero width bit fields is allowed and will have a length, in bytes, of two times the number of zero width bit fields.

For unions, there is one special case:

- Unions whose largest element is a bit field of width 16 or less have a size of 2 bytes. If the width of the bit field is greater than 16, the size of the union is 4 bytes.

Example

The following example uses these symbols to show padding and boundaries:

```
p = padding
| = halfword boundary
: = byte boundary
```

For:

```
#pragma options align=mac68k
struct A {
  char a;
  int : 0;
  int b : 4;
  int c : 17;
}

sizeof(A) == 8
```

The layout of A is:

```
|a:p|b .. :p|c:c|c .. :p|
```

Type Compatibility between RISC System/6000 and Macintosh Systems

Different aggregate types with identical members are not compatible. Therefore such aggregates cannot be assigned to each other.

RELATED REFERENCES

-qalign Compiler Option
RISC System/6000 Alignment Rules (page 14)
Packed Alignment Rules (page 12)
Alignment Rules for Nested Aggregates (page 12)

Alignment Rules for Nested Aggregates

Aggregates with different alignments can be nested. Each aggregate is laid out using the alignment rules applicable to it. The start position of the nested aggregate is determined by the alignment rules of the aggregate in which it is nested.

Example
The following example uses these symbols to show padding and boundaries:

```
p = padding
| = halfword boundary
: = byte boundary
```

For:

```
#pragma options align=mac68k
struct A {
  char a;
#pragma options align=power
  struct B {
    int b;
    char c;
  } B1;    // <-- B1 laid out using RISC System/6000 alignment rules
#pragma options align=reset    // <-- has no effect on A or B, but
                               //     on subsequent structs
  char d;
};

sizeof(A) == 12
```

The layout of A is:

```
|a:p|b:b|b:b|c:p|p:p|d:p|
```

RELATED REFERENCES
-qalign Compiler Option
RISC System/6000 Alignment Rules (page 14)
Macintosh and Twobyte Alignment Rules (page 10)
Packed Alignment Rules (page 12)

Packed Alignment Rules

All structures are byte-aligned regardless of their members. All members are also byte-aligned. Bit fields are byte-aligned, but bit-field members are not.

Example
The following example uses these symbols to show padding and boundaries:

```
p = padding
| = halfword boundary
: = byte boundary
```

For:

```
#pragma options align=packed
struct {
  char a;
  double b;
} B;
#pragma options align=reset

sizeof(B) == 9
```

The layout of B is:

```
|a:b|b:b|b:b|b:
```

Packed Bit Fields

The following rules apply when laying out packed bit fields.

- An individual bit field can be at most 32 bits long.
- Bit fields are packed together into the current word. If a bit field extends beyond the current word, it starts at the next byte boundary.
- A bit field of width zero causes the next class member to start at the next byte boundary. If the zero-width bit field is already at a byte boundary, the next structure member starts at this boundary.
- A nonbit field following a bit field is aligned on the next byte boundary.

Example

```
#pragma options align=packed
struct {
int a : 8;
int b : 10;
int c : 12;
int d : 4;
int e : 3;
int : 0;
int f : 1;
char g;
} A;
#pragma options align=reset
```

sizeof(A) == 7

The layout of A is:

Member Name	Displacement bytes (bits)
a	0
b	1
c	2 (2)
d	4
e	4 (4)
f	5
g	6

RELATED REFERENCES

Mapping of Fundamental Data Types (page 9)
Mapping of Compound Data Types (page 9)
-qalign Compiler Option
RISC System/6000 Alignment Rules (page 14)
Alignment Rules for Nested Aggregates (page 12)

RISC System/6000 Alignment Rules

RISC System/6000 alignment is the default setting for the -qalign compiler option. On the RISC System/6000 system, an aggregate is aligned according to its most strictly aligned member. Within aggregates, members are aligned according to their type. The table below summarizes size and alignment information for each type.

Type, Size, and Alignment for the RISC System/6000 System		
Type	**Alignment of Member**	**Size (bytes)**
char	byte aligned	1
short	halfword aligned	2
(long) int	word aligned in 32-bit mode 8-byte aligned in 64-bit mode	4 in 32-bit mode 8 in 64-bit mode
long long int	doubleword aligned	8
pointer	word aligned	4
float	word aligned	4
double	doubleword aligned if -qalign=natural is set. Otherwise, word aligned.	8
long double with the -qlongdouble compiler option.	long doubleword aligned if -qalign=natural is set. Otherwise, word aligned.	16

Notes:

1. The entire object is aligned on the same boundary as its most strictly aligned member.
2. Each member is assigned the lowest available offset with the appropriate alignment (internal padding).
3. The object's size is increased, if necessary, to make it a multiple of the size of its most strictly aligned member. (For example, if the object contains a word, it is padded to a word boundary.)

On the RISC System/6000 system, if a double is the first member of a struct, the struct is 8-byte (doubleword) aligned. If a long double is the first member of a struct, the struct is 16-byte aligned.

Bit Fields for RISC System/6000 Format

The following rules apply when you are laying out bit fields in structs:

- structs containing bit fields are 4-byte (word) aligned.
- Bit fields can be at most 32 bits long. (In 64-bit mode, long bit-fields can be at most 64 bits long).
- Bit fields are packed into the current word. If a bit field would cross a word boundary, it starts at the next word boundary.
- A bit field of width zero causes the bit field that immediately follows it to be aligned at the next word boundary. If the zero width bit field is at a word boundary, the next bit field starts at this boundary.
- A struct containing nothing but zero-width bit fields is allowed. In C it will have a length of 0 bytes. In C++ the struct will have a length of 8 bytes if the first such bit field is of type long or long long. Otherwise, in C++, it will have a length of 4 bytes.

In the C language, you can specify bit fields as char or short instead of int, but VisualAge C++ maps them as if they were unsigned int. In extended mode, you can use the sizeof operator on a bit field. (The sizeof operator on a bit field always returns 4.)

RELATED REFERENCES

-qalign Compiler Option
Macintosh and Twobyte Alignment Rules (page 10)
Packed Alignment Rules (page 12)
Alignment Rules for Nested Aggregates (page 12)

Storage of float and double Types

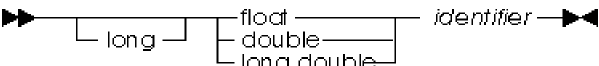
Specifier	Description
`float`	Allocates 4 bytes of data storage.
`double`	Allocates 8 bytes of data storage.
`long double`	Normally allocates 8 bytes of data storage in 32-bit compiler mode.

Notes:

1. The amount of storage allocated for a float, double, or long double floating-point variable is implementation-dependent. On all compilers, the storage size of a float variable is less than or equal to the storage size of a double variable.
2. In extended mode, the C compiler supports long float, but this is a non-portable language extension.

To declare a data object having a floating-point type, use the *float specifier*.

The float specifier has the form:



The declarator for a simple floating-point declaration is an identifier. You can initialize a simple floating-point variable with a float constant or with a variable or expression that evaluates to an integer or floating-point number. The storage class of a variable determines how you initialize the variable.

The following example defines the identifier `pi` as an object of type double:

```
double pi;
```

The following example defines the float variable `real_number` with the initial value 100.55:

```
static float real_number = 100.55f;
```

The following example defines the float variable `float_var` with the initial value 0.0143:

```
float float_var = 1.43e-2f;
```

The following example declares the long double variable `maximum`:

```
extern long double maximum;
```

The following example defines the array `table` with 20 elements of type double:

```
double table[20];
```

RELATED CONCEPTS

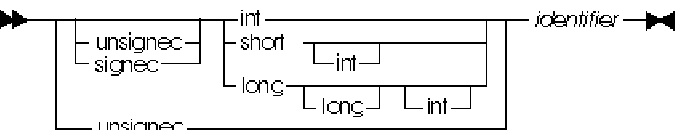
Data Mapping (page 8)

Storage of int, long, and short Types

Specifier	Description
`short, short int`	Allocates 2 bytes of data storage.
`int`	Allocates 4 bytes of data storage.
`long, long int`	Allocates 4 bytes of data storage in 32-bit compiler mode, and 8 bytes in 64-bit compiler mode.
`long long, long long int`	Allocates 8 bytes of data storage. The compiler supports **long long**, but this is not a standard data type. Though needed for some system programming, it may not be portable to other systems.
Notes: The amount of storage allocated for an int, short, or long integer variable is implementation-dependent.	

To declare a data object having an integer data type, use an int type specifier.

The **int** specifier has the form:



The declarator for a simple integer definition or declaration is an identifier. You can initialize a simple integer definition with an integer constant or with an expression that evaluates to a value that can be assigned to an integer. The storage class of a variable determines how you can initialize the variable.

The unsigned prefix indicates that the object is a nonnegative integer. Each unsigned type provides the same size storage as its signed equivalent. For example, int reserves the same storage as unsigned int. Because a signed type reserves a sign bit, an unsigned type can hold a larger positive integer than the equivalent signed type.

The following example defines the short int variable `flag`:

```
short int flag;
```

The following example defines the int variable `result`:

```
int result;
```

The following example defines the unsigned long int variable `ss_number` as having the initial value `438888834`:

```
unsigned long ss_number = 438888834ul;
```

The following example defines the identifier `sum` as an object of type int. The initial value of `sum` is the result of the expression `a + b`:

```
extern int a, b;
auto sum  = a + b;
```

RELATED CONCEPTS

Data Mapping (page 8)

Chapter 3. Signals and Exception Handling

Choose Signal Handlers

You can use your signal handlers and operating system signal handlers alone and in combination. Signal handlers can be complex to write and difficult to debug, but creating your own has these advantages:

- You receive additional information about the error condition.
- You can intercept any operating system signal, including those the VisualAge C++library passes back to the operating system because there is no C signal for handling them.

When Special Handling is Required
Floating point exceptions and two classes of library functions, math functions and critical functions, require special handling. Operating system signals that occur in all other library functions are treated as though they occurred in regular user code.

If your program links with shared libraries that link to more than one library environment, you must take steps to ensure that the right handler is called.

RELATED CONCEPTS
Signals and Exceptions (page 20)

Signal Handling

You can handle signals in either of the following ways:

- Accept the default handling provided by VisualAge C++, which usually results in program termination with a message.
- Program signal handling in C and C++ programs.

When to Simply Debug
To eliminate signals that you suspect are due to program logic, use the debugger. The debugger provides complete notification and stack tracing is available.

Here are some other common problems:

- Improper use of memory. Using a pointer to an object that has already been freed can cause an exception.
- Using an invalid pointer.
- Passing an invalid parameter to a system function.
- Return codes from library or system calls that are not checked.

RELATED CONCEPTS
Signals (page 20)

RELATED TASKS
Choose Signal Handlers (page 19)
Program Signal Handling (page 20)

© Copyright IBM Corp. 1998, 2002

RELATED REFERENCES
Signal Handling Considerations (page 21)

Signals

Signals are software interrupts for which the program can install custom interrupt handlers. Signal handlers provide both a way of dealing with exceptional error conditions (such as a divide by zero error) and as a primitive means of interprocess communication.

Signals are a facility built on operating exceptions compliant with the ANSI C standard. You can use them in C and C++ to intercept operating system exceptions in a portable way. Different signals differentiate between error conditions. The following kinds of events raise signals:

- A machine interrupt, such as divide by zero. This is a very common source of signals.
- Your program can send a signal to itself with the raise function.
- The shell can generate signals in response to user-defined keystrokes. For example, Ctrl-C is commonly defined as the SIGINT signal. Use the stty -a command to determine which signals are set for your shell.
- The operating system may send a signal. For example, SIGSEGV may be sent for an invalid memory reference.

Operating system signals are synchronous or asynchronous depending on the relationship between their cause and the execution of the program.

- Synchronous signals are caused by code in the thread that receives the signal. Most operating system are synchronous.
- Asynchronous signals are caused by actions outside of your current thread, for example, typing Ctrl-C.

C++ **C++ Exception Handling**
C++ exceptions constructs, such as try, throw and catch, exist only within the C++ language. However, C++ exception handlers cannot intercept operating system exceptions, such as access violations.

RELATED CONCEPTS
Signal Handling (page 19)

RELATED TASKS
Choose Signal Handlers (page 19)
Program Signal Handling (page 20)

Program Signal Handling

Use the signal function to specify how to handle signals. For each signal, you can specify one of the types of handlers listed below. The signal constants are defined in <signal.h>.

1. SIG_DFL
   Specifies the default action. This is the initial setting for all signals. For most signals, the default action is to terminate the process with an error message.
2. SIG_IGN
   Ignores the condition and tries to continue running the program. If you specify

SIG_IGN for a signal that cannot be ignored, such as division by zero, the VisualAge C++library treats the signal as if SIG_DFL was specified.

3. Your own signal handler function
   Registers the function you specify. This can be a function you have written. When the signal is reported and your function is called, signal handling is reset to SIG_DFL to prevent recursion should the same signal be reported from your function.

To reset default handling for a signal, call the signal in a statement similar to the following. Specify the signal name in the first argument of signal.

```
signal(name, SIG_DFL);
```

RELATED CONCEPTS

Signals (page 20)
Signal Handling (page 19)

Signal Handling Considerations

Considerations when *Registering* a Signal Handler:

- You can register any function as a signal handler. Make sure you are registering a valid function.
- If your signal handler resides in a shared library, ensure that you change the signal handler when you unload the library. You will see no warnings or error messages if you unload the library without changing the handler, but your program will probably terminate the next time the handler is called. If another shared library has been loaded in the same address range, your program may continue but with undefined results.
- The SIGILL signal is not guaranteed to occur when you call an invalid function using a pointer. If the pointer points to a valid instruction stream, SIGILL is not raised.

Considerations when *Writing* a Signal Handler:

- Your signal handler should not assume that SIGSEGV always implies an invalid data pointer. Other events, such as an address pointer to outside of your code segment, can raise this signal.
- The SIGILL signal is not guaranteed to occur when you call an invalid function using a pointer. If the pointer points to a valid instruction stream, SIGILL is not raised.
- When you call longjmp to leave a signal handler, be sure that the buffer you are jumping to was created by the thread that you are in. Do not call setjmp from one thread and longjmp from another. The VisualAge C++library terminates a process where such a call is made.
- Declare variables referenced by both the signal handler and by other code to be volatile, to ensure they are always updated when they are referenced.
- C++ The ANSI C++ standard does not recommend mixing exception handling and signal handling.
- If a file I/O routine from <stdio.h> is taking input from the console when an asynchronous signal (SIGINT or SIGTERM) occurs, the behavior depends on whether a signal handler is registered. If no handler is registered, the input stream is terminated and the default signal action is taken. If a handler is registered, entry to the handler is deferred until the library routine returns.

- C++ Do not use a C++ throw to exit an asynchronous signal handler.

RELATED CONCEPTS

Signals (page 20)
Signal Handling (page 19)

RELATED REFERENCES

Example of Volatile Variables (page 22)

Example of Using Volatile Variables

User variables that are referenced by multiple threads should have the attribute **volatile**to ensure that all changes to the value of the variable are performed immediately by the compiler.

Because of the way the VisualAge C++ optimizes code, the following example may not work as intended if it is built with the optimization options.

```
#include <io.h>
#include <signal.h>
#include <stdio.h>
#include <string.h>

void sig_handler(int);
static int stepnum;

int main()
{
   stepnum = 0;
   signal(SIGSEGV, sig_handler);
   /* code omitted - does not use stepnum */
   stepnum = 1;
   /* code omitted - does not use stepnum */
   stepnum = 2
   return 0;
}

void sig_handler(int x)
{
   char FileData[50];
   sprintf(FileData, "Error at Step %d\n\r", stepnum);
   write (2, FileData, strlen(fileData));
}
```

An optimized program may not immediately store the value 1 when 1 is assigned to the variable stepnum. It may never store the value 1 and only store the value 2. If a signal occurs between the assignments to stepnum, the value passed to *sig_handler* may not be correct.

Declaring a variable (*stepnum*) as volatile indicates to the compiler that references to the variable have side effects, or that the variable may change in ways the compiler cannot determine. Optimization will not eliminate any action involving the volatile variable. Changes to the value of the variable are then stored immediately, and uses of the variable will always cause it to be re-fetched from memory.

RELATED CONCEPTS

Signals (page 20)
Signal Handling (page 19)

RELATED TASKS

Program Signal Handling (page 20)

RELATED REFERENCES

Variables and Optimization (page 54)

Chapter 4. Using Memory Heaps

Memory Management Functions

The memory management functions defined by ANSI are `calloc`, `malloc`, `realloc`, and `free`. These regular functions allocate and free memory from the default runtime heap. VisualAge C++ includes another function, `_heapmin`, to return unused memory to the system. VisualAge C++ also provides enhanced versions of memory management functions that can help you improve program performance (link to the libhm.a library), work with user heaps, or debug your programs.

All the versions actually work the same way. They differ only in what heap they allocate from, and in whether they save information to help you debug memory problems. The memory allocated by all of these functions is suitably aligned for storing any type of object.

The table below summarizes the different versions of memory management functions, using `malloc`as an example of how the names of the functions change for each version.

	Regular Version	Debug Version
Default Heap	`malloc`	`_debug_malloc`
User-Created Heap	`_umalloc`	`_debug_umalloc`

Heap-Specific Functions
Use heap-specific versions of memory allocation functions to allocate and free memory from user-created heaps that you specify. If you want, you can also explicitly specify the runtime heap. The names of user-created heaps are prefixed by `_u`(for "user heaps"), for example, `_umalloc`, and they are defined in `<umalloc.h>`.

When working with user-created heaps, you need to link to the libhu.a library. Heap-specific functions provided in this library are:
- `_ucalloc`
- `_umalloc`
- `_uheapmin`

There are no heap-specific versions of `realloc` or `free`. These standard functions always determine which heap memory is allocated from, and can be used with both user-created and runtime memory heaps.

Debug Functions
Use these functions to allocate and free memory from the default runtime heap, just as you would use the regular versions. They also provide information that you can use to debug memory problems.

Use the -qheapdebug compiler option to automatically map all calls to the regular memory management functions to their debug versions. You can also call the debug versions explicitly. **Note:** do not use the -brtl option with -qheapdebug.

© Copyright IBM Corp. 1998, 2002

Note: If you parenthesize the calls to the regular memory management functions, they are *not* mapped to their debug versions.

You should place a #pragma strings(readonly) directive at the top of each source file that will call debug functions, or in a common header file that each includes. This directive is not essential, but it ensures that the file name passed to the debug functions can't be overwritten, and that only one copy of the file name string is included in the object module.

The names of the debug versions are prefixed by `_debug_`, for example, `_debug_malloc`, and they are defined in `<malloc.h>`and `<stdlib.h>`.

The functions provided are:
- `_debug_calloc`
- `_debug_free`
- `_debug_heapmin`
- `_debug_malloc`
- `_debug_realloc`

The debug_malloc, debug_realloc, and debug_free functions set the memory areas they affect to a specific, repeating fill pattern.

In addition to their usual behavior, these functions also store information (file name and line number) about each call made to them. Each call also automatically checks the heap by calling `_heap_check` (described below).

Three additional debug memory management functions do not have regular counterparts:
- `_dump_allocated`

  Prints information to **stderr** about each memory block currently allocated by the debug functions.
- `_dump_allocated_delta`

  Prints information to file handle 2 about each memory block allocated by the debug functions since the last call to `_dump_allocated`or `_dump_allocated_delta`.
- `_heap_check`

  Checks all memory blocks allocated or freed by the debug functions to make sure that no overwriting has occurred outside the bounds of allocated blocks or in a free memory block.

The debug functions call `_heap_check`automatically; and you can also call this function explicitly. The `_dump_allocated` and `_dump_allocated_delta`functions must be explicitly called.

Heap-Specific Debug Functions

The heap-specific functions also have debug versions that work just like the regular debug versions. Use these functions to allocate and free memory from the user-created heap you specify, and also provide information that you can use to debug memory problems in your own heaps.

Use the -qheapdebug compiler option to automatically map all calls to the regular memory management functions to their debug versions. You can also call the debug versions explicitly.

Note: If you parenthesize the calls to the regular memory management functions, they are **not** mapped to their debug versions.

The names of the heap-specific debug versions are prefixed by `_debug_u`, for example, `_debug_umalloc`, and they are defined in `<umalloc.h>`.

The functions provided are:
- `_debug_ucalloc`
- `_debug_uheapmin`
- `_debug_umalloc`
- `_udump_allocated`
- `_udump_allocated_delta`
- `_uheap_check`

The debug_umalloc function sets the memory areas they affect to a specific, repeating fill pattern.

There are no heap-specific debug versions of `_debug_realloc` or `_debug_free`. These functions always determine which heap memory is allocated from, and can be used with both user-created and runtime memory heaps.

RELATED CONCEPTS

Managing Memory with Multiple Memory Heaps (page 27)
Types of Memory (page 29)
Debugging Memory Heaps (page 29)

RELATED TASKS

Create and Use a Fixed Size Heap (page 31)
Create and Use an Expandable Heap (page 33)
Debug Programs with Heap Memory (page 35)
Change the Default Heap Used in a Program (page 37)

RELATED REFERENCES

Example of Creating and Using a User Heap (page 38)
Example of Creating and Using a Shared-Memory User Heap (page 39)

Managing Memory with Multiple Heaps

VisualAge C++ lets you create and use your own pools of memory, called *heaps*. You can use your own heaps in place of or in addition to the default VisualAge C++ runtime heap to improve the performance of your program.

Using your own heaps is entirely optional, and your applications will work perfectly well using the default memory management provided (and used by) the VisualAge C++ run-time library. If you want to improve the performance and memory management of your program, multiple heaps can help you. Otherwise, you can ignore this section and any heap-specific library functions.

Why Use Multiple Heaps?
Using a single runtime heap is fine for most programs. However, using multiple

heaps can be more efficient and can help you improve your program's performance and reduce wasted memory for a number of reasons:

- When you allocate from a single heap, you may end up with memory blocks on different pages of memory. For example, you might have a linked list that allocates memory each time you add a node to the list. If you allocate memory for other data in between adding nodes, the memory blocks for the nodes could end up on many different pages. To access the data in the list, the system may have to swap many pages, which can significantly slow your program.

  With multiple heaps, you can specify which heap you allocate from. For example, you might create a heap specifically for the linked list. The list's memory blocks and the data they contain would remain close together on fewer pages, reducing the amount of swapping required.
- In multithread applications, only one thread can access the heap at a time to ensure memory is safely allocated and freed. For example, say thread 1 is allocating memory, and thread 2 has a call to `free`. Thread 2 must wait until thread 1 has finished its allocation before it can access the heap. Again, this can slow down performance, especially if your program does a lot of memory operations.

  If you create a separate heap for each thread, you can allocate from them concurrently, eliminating both the waiting period and the overhead required to serialize access to the heap.
- With a single heap, you must explicitly free each block that you allocate. If you have a linked list that allocates memory for each node, you have to traverse the entire list and free each block individually, which can take some time.

  If you create a separate heap for that linked list, you can destroy it with a single call and free all the memory at once.
- When you have only one heap, all components share it (including the VisualAge C++ runtime library, vendor libraries, and your own code). If one component corrupts the heap, another component might fail. You may have trouble discovering the cause of the problem and where the heap was damaged.

  With multiple heaps, you can create a separate heap for each component, so if one damages the heap (for example, by using a freed pointer), the others can continue unaffected. You also know where to look to correct the problem.

You can create heaps of regular memory or shared memory, and you can have any number of heaps of any type. The only limit is the space available on your operating system (your machine's memory and swapper size, minus the memory required by other running applications).

VisualAge C++ provides heap-specific versions of the memory management functions, for example, `umalloc`and so on. Debug versions of all memory management functions are provided, including the heap-specific ones. VisualAge C++ also provides additional functions that you can use to create and manage your own heaps of memory, such as udefault.

RELATED CONCEPTS

Memory Management Functions (page 25)
Types of Memory (page 29)
Debugging Memory Heaps (page 29)

RELATED TASKS

Create and Using a Fixed Size Heap (page 31)
Create and Using an Expandable Heap (page 33)

Debug Programs with Heap Memory (page 35)
Change the Default Heap Used in a Program (page 37)

RELATED REFERENCES

Example of Creating and Using a User Heap (page 38)
Example of Creating and Using a Shared-Memory User Heap (page 39)

Types of Memory

There are two types of memory:

1. Regular memory

   Most programs use regular memory. This is the type provided by the default runtime heap.

2. Shared memory

   Heaps of shared memory can be shared between processes or applications. If you want other processes to use the heaps you have created, you must pass them the heap handle and give them access to the heap. Use `_ucreate`to create the heap.

RELATED CONCEPTS

Memory Management Functions (page 25)
Managing Memory with Multiple Memory Heaps (page 27)
Debugging Memory Heaps (page 29)

RELATED REFERENCES

Example of Creating and Using a User Heap (page 38)
Example of Creating and Using a Shared-Memory User Heap (page 39)

Debugging Memory Heaps

VisualAge C++ provides two sets of functions for debugging memory problems:

1. Debug versions of all memory management functions
2. Heap-checking functions similar to those provided by other compilers.

Debug Memory Management Functions
Debug versions of the heap-specific memory management functions are provided, just as they are for the regular versions. Each debug version performs the same function as its non-debug counterpart. In addition, the debug version calls `_uheap_check`to check the heap used in the call, and records the file and line number where the memory was allocated or freed. You can then use `_dump_allocated` or `_dump_allocated_delta`to display information about currently allocated memory blocks. Information is printed to stderr.

You can use debug memory management functions for any type of heap, including shared memory. To use the debug versions, specify the -qheapdebug compiler option. VisualAge C++ then maps all calls to memory management functions (regular or heap-specific) to the corresponding debug versions.

Note: If you parenthesize the name of a memory management function, the function is *not* mapped to the debug version.

Heap-Checking Functions
VisualAge C++also provides some functions for validating user heaps: _uheapchk, _uheapset, and _uheap_walk. Each of these functions also has a non-heap-specific version that validates the default heap.

Both _uheapchk and _uheapsetcheck the specified heap for minimal consistency; _uheapchkchecks the entire heap, while _uheapset checks only the free memory. _uheapsetalso sets the free memory in the heap to a value you specify. _uheap_walktraverses the heap and provides information about each allocated or freed object to a callback function that you provide. You can then use the information however you like.

These heap-checking functions are defined in **<umalloc.h>** (the regular versions are also in <malloc.h>). They are not controlled by a compiler option, so you can use them in your program at any time.

Which Should I Use?
Both sets of debugging functions have their benefits and drawbacks. Which you choose to use depends on your program, your problems, and your preference.

The debug memory management functions provide detailed information about all allocation requests you make with them in your program. You don't need to change any code to use the debug versions; you need only specify the -qheapdebug option.

On the other hand, the heap-checking functions perform more general checks on the heap at specific points in your program. You have greater control over where the checks the occur. The heap-checking functions also provide compatibility with other compilers that offer these functions. You only have to rebuild the modules that contain the heap-checking calls. However, you have to change your source code to include these calls, which you will probably want to remove in your final code. Also, the heap-checking functions only tell you if the heap is consistent or not; they do not provide the details that the debug memory management functions do.

What you may choose to do is add calls to heap-checking functions in places you suspect possible memory problems. If the heap turns out to be corrupted, you may want to rebuild with -qheapdebug.

Note: When the debug memory option -qheapdebug is specified, code is generated to *pre-initialize* the local variables for all functions. This makes it much more likely that uninitialized local variables will be found during the normal debug cycle rather than much later (usually when the code is optimized).

Regardless of which debugging functions you choose, your program requires additional memory to maintain internal information for these functions. If you are using fixed-size heaps, you may have to increase the heap size in order to use the debugging functions.

RELATED CONCEPTS
Memory Management Functions (page 25)
Managing Memory with Multiple Memory Heaps (page 27)
Types of Memory (page 29)

RELATED TASKS
Create and Use a Fixed Size Heap (page 31)

Create and Use an Expandable Heap (page 33)
Debug Programs with Heap Memory (page 35)
Change the Default Heap Used in a Program (page 37)

RELATED REFERENCES

heapdebug Compiler Option
_debug_calloc - Allocate and Initialize Memory (page 265)
_debug_free - Free Allocated Memory (page 266)
_debug_heapmin - Free Unused Memory in the Default Heap (page 268)
_debug_malloc - Allocate Memory (page 269)
_debug_memcpy - Copy Bytes (page 271)
_debug_memmove - Copy Bytes (page 273)
_debug_memset - Set Bytes to Value (page 274)
_debug_realloc - Reallocate Memory Block (page 276)
_debug_strcat - Concatenate Strings (page 278)
_debug_strcpy - Copy Strings (page 279)
_debug_strncat - Concatenate Strings (page 282)
_debug_strncpy - Copy Strings (page 284)
_debug_strnset - Set Characters in String (page 281)
_debug_strset - Set Characters in String (page 285)
_debug_ucalloc - Reserve and Initialize Memory from User Heap (page 287)
_debug_uheapmin - Free Unused Memory in User Heap (page 289)
_debug_umalloc - Reserve Memory Block from User Heap (page 290)
Example of Creating and Using a User Heap (page 38)
Example of Creating and Using a Shared-Memory User Heap (page 39)

Create and Use a Fixed Size Heap

Before creating a heap, you must first allocate a block of memory large enough to hold the heap. The block must be large enough to satisfy all the memory requests your program will make of it, and also be able to hold internal information required to manage the heap. Once the block is fully allocated, further allocation requests to the heap will fail.

The internal information requires `_HEAP_MIN_SIZE` bytes (`_HEAP_MIN_SIZE` is defined in <umalloc.h>). You cannot create a heap smaller than this. Add the amount of memory your program requires to this value to determine the size of the block you need to get. Also, make sure the block is the correct type (regular or shared) for the heap you are creating.

After you have allocated a block of memory, create the heap with `_ucreate`.

For example:

```
Heap_t fixedHeap;    /* this is the "heap handle" */
/* get memory for internal info plus 5000 bytes for the heap */
static char block[_HEAP_MIN_SIZE + 5000];

fixedHeap = _ucreate(block, (_HEAP_MIN_SIZE+5000),  /* block to use */
                    !_BLOCK_CLEAN,   /* memory is not set to 0   */
                    _HEAP_REGULAR,   /* regular memory           */
                    NULL, NULL);     /* we'll explain this later */
```

The `!_BLOCK_CLEAN` parameter indicates that the memory in the block has not been initialized to 0. If it were set to 0 (for example, by `memset`), you would specify `_BLOCK_CLEAN`. The `calloc` and `_ucalloc`functions use this information to improve their efficiency; if the memory is already initialized to 0, they don't need to initialize it.

The fourth parameter indicates what type of memory the heap contains: regular (_HEAP_REGULAR) or shared (_HEAP_SHARED). The different memory types are described in Types of Memory. (page 29)

For a fixed-size heap, the last two parameters are always NULL.

Use Your Heap
Once you have created your heap, you can open it for use by calling _uopen:

```
_uopen(fixedHeap);
```

This opens the heap for that particular process; if the heap is shared, each process that uses the heap needs its own call to _uopen.

You can then allocate and free from your own heap just as you would from the default heap. To allocate memory, use _ucalloc or _umalloc. These functions work just like calloc and malloc, except you specify the heap to use as well as the size of block that you want. For example, to allocate 1000 bytes from fixedHeap:

```
void *up;
up = _umalloc(fixedHeap, 1000);
```

To reallocate and free memory, use the regular realloc and freefunctions. Both of these functions always check what heap the memory came from, so you don't need to specify the heap to use. For example, the realloc and freecalls in the following code fragment look exactly the same for both the default heap and your heap:

```
void *p, *up;
p = malloc(1000);   /* allocate 1000 bytes from default heap */
up = _umalloc(fixedHeap, 1000);  /* allocate 1000 from fixedHeap */

realloc(p, 2000);   /* reallocate from default heap */
realloc(up, 100);   /* reallocate from fixedHeap    */

free(p);            /* free memory back to default heap */
free(up);           /* free memory back to fixedHeap    */
```

For any object, you can find out what heap it was allocated from by calling _mheap. You can also get information about the heap itself by calling _ustats, which tells you:

- How much memory the heap holds (excluding memory used for overhead)
- How much memory is currently allocated from the heap
- What type of memory is in the heap
- The size of the largest contiguous piece of memory available from the heap

When you call any heap function, make sure the heap you specify is valid. If the heap is not valid, the behavior of the heap functions is undefined.

Add to a Fixed-Size Heap
Although you created the heap with a fixed size, you can add blocks of memory to it with _uaddmem. This can be useful if you have a large amount of memory that is allocated conditionally. Like the starting block, you must first allocate memory for a block of memory. This block will be added to the current heap, so make sure the block you add is the same type of memory as the heap you are adding it to.

For example, to add 64K to fixedHeap:

```
static char newblock[65536];

_uaddmem(fixedHeap,       /* heap to add to   */
         newblock, 65536, /* block to add     */
         _BLOCK_CLEAN);   /* sets memory to 0 */
```

Using `_uaddmem` is the only way to increase the size of a fixed heap.

Note: For every block of memory you add, a small number of bytes from it are used to store internal information. To reduce the total amount of overhead, it is better to add a few large blocks of memory than many small blocks.

Destroy Your Heap
When you have finished using the heap, close it with `_uclose`. Once you have closed the heap in a process, that process can no longer allocate from or return memory to that heap. If other processes share the heap, they can still use it until you close it in each of them. Performing operations on a heap after you've closed it causes undefined behavior.

To finally destroy the heap, call `_udestroy`. If blocks of memory are still allocated somewhere, you can force the destruction. Destroying a heap removes it entirely even if it was shared by other processes. Again, performing operations on a heap after you've destroyed it causes undefined behavior.

After you destroy your fixed-size heap, it is up to you to return the memory for the heap (the initial block of memory you supplied to `_ucreate`and any other blocks added by `_uaddmem`) to the system.

RELATED CONCEPTS
Memory Management Functions (page 25)
Managing Memory with Multiple Memory Heaps (page 27)
Types of Memory (page 29)
Debugging Memory Heaps (page 29)

RELATED TASKS
Create and Use an Expandable Heap (page 33)
Debug Programs with Heap Memory (page 35)
Change the Default Heap Used in a Program (page 37)

RELATED REFERENCES
Example of Creating and Using a User Heap (page 38)
Example of Creating and Using a Shared-Memory User Heap (page 39)

Create and Use an Expandable Heap

When using a fixed-size heap, the initial block of memory must be large enough to satisfy all allocation requests made to it. You can also, however, create a heap that can expand and contract as your program needs demand.

With the VisualAge C++ runtime heap, when not enough storage is available for your `malloc`request, the runtime gets additional storage from the system. Similarly, when you minimize the heap with `_heapmin`or when your program ends, the runtime returns the memory to the operating system.

When you create an expandable heap, you provide your own functions to do this work (we'll call them `getmore_fn` and `release_fn`, although you can name them

whatever you choose). You specify pointers to these functions as the last two parameters to _ucreate (instead of the NULL pointers you used to create a fixed-size heap). For example:

```
Heap_t growHeap;
static char block[_HEAP_MIN_SIZE];  /* get block */

growHeap = _ucreate(block, _HEAP_MIN_SIZE,   /* starting block */
                    !_BLOCK_CLEAN,      /* memory not set to 0 */
                    _HEAP_REGULAR,      /* regular memory      */
                    getmore_fn,     /* function to expand heap */
                    release_fn);    /* function to shrink heap */
```

Note: You can use the same getmore_fn and release_fn for more than one heap, as long as the heaps use the same type of memory and your functions are not written specifically for one heap.

Expand Your Heap

When you call _umalloc(or a similar function) for your heap, _umalloctries to allocate the memory from the initial block you provided to _ucreate. If not enough memory is there, it then calls your getmore_fn. Your getmore_fn then gets more memory from the operating system and adds it to the heap. It is up to you how you do this.

Your getmore_fn must have the following prototype:

```
void *(*getmore_fn)(Heap_t uh, size_t *size, int *clean);
```

The *uh* is the heap to be expanded.

The *size* is the size of the allocation request passed by _umalloc. You probably want to return enough memory at a time to satisfy several allocations; otherwise every subsequent allocation has to call getmore_fn, reducing your program's execution speed. Make sure that you update the *size* parameter. if you return more than the *size* requested.

Your function must also set the clean parameter to either _BLOCK_CLEAN, to indicate the memory has been set to 0, or !_BLOCK_CLEAN, to indicate that the memory has not been initialized.

The following fragment shows an example of a getmore_fn:

```
static void *getmore_fn(Heap_t uh, size_t *length, int *clean)
{
   char *newblock;

   /* round the size up to a multiple of 64K */
   *length = (*length / 65536) * 65536 + 65536;

   *clean = _BLOCK_CLEAN;  /* mark the block as "clean" */
   return(newblock);       /* return new memory block   */
}
```

Be sure that your getmore_fn allocates the right type of memory (regular or shared) for the heap. There are also special considerations for shared memory, as described in Types of Memory (page 29).

You can also use _uaddmemto add blocks to your heap, as you did for the fixed heap in Create and Use a Fixed-Size Heap (page 31). _uaddmemworks exactly the same way for expandable heaps.

Shrink Your Heap
To coalesce the heap (return all blocks in the heap that are totally free to the system), use _uheapmin. _uheapmin works like _heapmin, except that you specify the heap to use.

When you call _uheapmin to coalesce the heap or _udestroyto destroy it, these functions call your *release_fn* to return the memory to the system. Again, it is up to you how you implement this function.

Your *release_fn* must have the following prototype:

```
void (*release_fn)(Heap_t uh, void *block, size_t size);
```

Where *uh* identifies the heap to be shrunk. The pointer *block* and its *size* are passed to your function by _uheapmin or _udestroy. Your function must return the memory pointed to by *block* to the system. For example:

```
static void release_fn(Heap_t uh, void *block, size_t size)
{
   free(block);
   return;
}
```

Notes:

1. _udestroycalls your release_fn to return all memory added to the *uh* heap by your *getmore_fn* or by _uaddmem. However, you are responsible for returning the initial block of memory that you supplied to _ucreate.
2. Because a fixed-size heap has no release_fn, _uheapminand _udestroy work slightly differently. Calling _uheapminfor a fixed-size heap has no effect but does not cause an error; _uheapminsimply returns 0. Calling _udestroyfor a fixed-size heap marks the heap as destroyed, so no further operations can be performed on it, but returns no memory. It is up to you to return the heap's memory to the system.

RELATED CONCEPTS

Memory Management Functions (page 25)
Managing Memory with Multiple Memory Heaps (page 27)
Types of Memory (page 29)
Debugging Memory Heaps (page 29)

RELATED TASKS

Create and Use a Fixed Size Heap (page 31)
Debug Programs with Heap Memory (page 35)
Change the Default Heap Used in a Program (page 37)
Example of Creating and Using a User Heap (page 38)
Example of Creating and Using a Shared-Memory User Heap (page 39)

Debug Programs with Heap Memory

VisualAge C++ provides debug versions of both general memory management functions and heap-specific memory management functions. To automatically call the debug versions of these functions, specify the -qheapdebug compiler option when compiling your program. Bear in mind that specifying this option can significantly increase the memory requirements and running time of your program.

Memory Allocation Fill Pattern
Some debug functions set all the memory they allocate to a specified fill pattern. This lets you easily locate areas in memory that your program uses.

The debug_malloc, debug_realloc, and debug_umalloc functions sets allocated memory to a default repeating `0xAA` fill pattern. To enable this fill pattern, export the `HD_FILL` environment variable.

The debug_free function sets all free memory to a repeating `0xFB` fill pattern.

Skip Heap Checks
Each debug function calls `_heap_check` (or `_uheap_check`) to check the heap. Although this is useful, it can also increase your program's memory requirements and decrease its execution speed.

To reduce the overhead of checking the heap on every debug memory management function, you can control how often the functions check the heap with the `HD_SKIP` environment variable. You will not need to do this for most of your applications unless the application is extremely memory intensive.

Set `HD_SKIP` like any other environment variable. The syntax for `HD_SKIP` is:

```
set HD_SKIP=increment,[start]
```

where:

increment	Specifies how often you want the debug functions to check the heap.
start	Optional. Use this parameter to start skipping heap checks after *start* calls to debug functions.

Note: The comma separating the parameters is optional.

When you use the *start* parameter to start skipping heap checks, you are trading off heap checks that are done implicitly against program execution speed. You should therefore start with a small increment (like 5) and slowly increase until the application is usable.

For example, if you specify:

```
set HD_SKIP=10
```

then every tenth debug memory function call performs a heap check. If you specify:

```
set HD_SKIP=5,100
```

then after 100 debug memory function calls, only every fifth call performs a heap check. Other than the heap check, the debug functions behave exactly the same as usual.

Use Stack Traces
Stack contents are traced for each allocated memory object. If the contents of an object's stack change, the traced contents are dumped.

The trace size is controlled by the `HD_STACK` environment variable. If this variable is not set, the compiler assumes a stack size of 10. To disable stack tracing, set the `HD_STACK` environment variable to 0.

RELATED CONCEPTS
Memory Management Functions (page 25)

Managing Memory with Multiple Memory Heaps (page 27)
Types of Memory (page 29)
Debugging Memory Heaps (page 29)

RELATED TASKS

Create and Use a Fixed Size Heap (page 31)
Create and Use an Expandable Heap (page 33)
Change the Default Heap Used in a Program (page 37)
Example of Creating and Using a User Heap (page 38)
Example of Creating and Using a Shared-Memory User Heap (page 39)

RELATED REFERENCES

heapdebug Compiler Option
_debug_calloc - Allocate and Initialize Memory (page 265)
_debug_free - Free Allocated Memory (page 266)
_debug_heapmin - Free Unused Memory in the Default Heap (page 268)
_debug_malloc - Allocate Memory (page 269)
_debug_memcpy - Copy Bytes (page 271)
_debug_memmove - Copy Bytes (page 273)
_debug_memset - Set Bytes to Value (page 274)
_debug_realloc - Reallocate Memory Block (page 276)
_debug_strcat - Concatenate Strings (page 278)
_debug_strcpy - Copy Strings (page 279)
_debug_strncat - Concatenate Strings (page 282)
_debug_strncpy - Copy Strings (page 284)
_debug_strnset - Set Characters in String (page 281)
_debug_strset - Set Characters in String (page 285)
_debug_ucalloc - Reserve and Initialize Memory from User Heap (page 287)
_debug_uheapmin - Free Unused Memory in User Heap (page 289)
_debug_umalloc - Reserve Memory Block from User Heap (page 290)

Change the Default Heap Used in a Program

The regular memory management functions (malloc and so on) always use whatever heap is currently the default for that thread. The initial default heap for all VisualAge C++ applications is the runtime heap provided by VisualAge C++. However, you can make your own heap the default by calling _udefault. Then all calls to the regular memory management functions allocate from your heap instead of the runtime heap.

The default heap changes only for the thread where you call _udefault. You can use a different default heap for each thread of your program if you choose.

This is useful when you want a component (such as a vendor library) to use a heap other than the VisualAge C++ runtime heap, but you can't actually alter the source code to use heap-specific calls. For example, if you set the default heap to a shared heap then call a library function that calls malloc, the library allocates storage in shared memory.

Because _udefault returns the current default heap, you can save the return value and later use it to restore the default heap you replaced. You can also change the default back to the VisualAge C++runtime heap by calling _udefault and specifying `_RUNTIME_HEAP` (defined in <umalloc.h>). You can also use this macro with any of the heap-specific functions to explicitly allocate from the runtime heap.

RELATED CONCEPTS

Memory Management Functions (page 25)
Managing Memory with Multiple Memory Heaps (page 27)
Types of Memory (page 29)
Debugging Memory Heaps (page 29)

RELATED TASKS

Create and Use a Fixed Size Heap (page 31)
Create and Use an Expandable Heap (page 33)
Debug Programs with Heap Memory (page 35)
Example of Creating and Using a User Heap (page 38)
Example of Creating and Using a Shared-Memory User Heap (page 39)

Example of Creating and Using a User Heap

The program below shows how you might create and use a heap.

Compile with -qheapdebug to map memory management functions to their debug versions.

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

static void *get_fn(Heap_t usrheap, size_t *length, int *clean)
{
   void *p;

   /* Round up to the next chunk size */
   *length = ((*length) / 65536) * 65536 + 65536;
   *clean = _BLOCK_CLEAN;
   p = calloc(*length,1);
   return (p);
}

static void release_fn(Heap_t usrheap, void *p, size_t size)
{
   free( p );
   return;
}

int main(void)
{
   void    *initial_block;
   long   rc;
   Heap_t  myheap;
   char    *ptr;
   int     initial_sz;

   /* Get initial area to start heap */
   initial_sz = 65536;
   initial_block = malloc(initial_sz);
   if(initial_block == NULL) return (1);

   /* create a user heap */
   myheap = _ucreate(initial_block, initial_sz, _BLOCK_CLEAN,
                     _HEAP_REGULAR, get_fn, release_fn);
   if (myheap == NULL) return(2);


   /* allocate from user heap and cause it to grow */
   ptr = _umalloc(myheap, 100000);
   _ufree(ptr);
```

```
   /* destroy user heap */
   if (_udestroy(myheap, _FORCE)) return(3);

   /* return initial block used to create heap */

   free(initial_block);
   return 0;
}
```

RELATED CONCEPTS

Memory Management Functions (page 25)
Managing Memory with Multiple Memory Heaps (page 27)
Types of Memory (page 29)
Debugging Memory Heaps (page 29)

RELATED TASKS

Create and Use a Fixed Size Heap (page 31)
Create and Use an Expandable Heap (page 33)
Debug Programs with Heap Memory (page 35)
Change the Default Heap Used in a Program (page 37)
Example of Creating and Using a Shared-Memory User Heap (page 39)

Example of Creating and Using a Shared-Memory User Heap

The following program shows how you might implement a heap shared between a parent and several child processes.

Example of a User Heap - Parent Process (page 39) shows the parent process, which creates the shared heap. First the main program calls the init function to allocate shared memory from the operating system (using CreateFileMapping) and name the memory so that other processes can use it by name. The init function then creates and opens the heap. The loop in the main program performs operations on the heap, and also starts other processes. The program then calls the term function to close and destroy the heap.

Example of a Shared User Heap - Child Process (page 42) shows the process started by the loop in the parent process. This process uses OpenFileMapping to access the shared memory by name, then extracts the heap handle for the heap created by the parent process. The process then opens the heap, makes it the default heap, and performs some operations on it in the loop. After the loop, the process replaces the old default heap, closes the user heap, and ends.

Example of a User Heap - Parent Process

```
/* The following program shows how you might implement
 a heap shared between a parent and several child processes.

 Example of a Shared User Heap - Parent Process shows the parent
 process, which creates the shared heap. First the main program
 calls the init function to allocate shared memory from the
 operating system (using CreateFileMapping) and name the memory
 so that other processes can use it by name. The init function
 then creates and opens the heap. The loop in the main program
 performs operations on the heap, and also starts other processes.
 The program then calls the term function to close and destroy the heap.

 */

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

```
#include <string.h>

#define PAGING_FILE  0xFFFFFFFF
#define MEMORY_SIZE  65536
#define BASE_MEM     (VOID*)0x01000000

static HANDLE hFile;      /* Handle to memory file                        */
static void*  hMap;       /* Handle to allocated memory                   */

typedef struct mem_info {
   void  * pBase;
   Heap_t  pHeap;
} MEM_INFO_T;

/*--------------------------------------------------------------------------*/
/* inithp:                                                                  */
/* Function to create and open the heap with a named shared memory object */
/*--------------------------------------------------------------------------*/
static Heap_t inithp(size_t heap_size)
{
   MEM_INFO_T info;                       /* Info structure               */

   /* Allocate shared memory from the system by creating a shared memory  */
   /* pool basing it out of the system paging (swapper) file.             */

   hFile = CreateFileMapping( (HANDLE) PAGING_FILE,
                              NULL,
                              PAGE_READWRITE,
                              0,
                              heap_size + sizeof(Heap_t),
                              "MYNAME_SHAREMEM" );
   if (hFile == NULL) {
     return NULL;
   }

   /* Map the file to this process' address space, starting at an address  */
   /* that should also be available in child processe(s)                   */

   hMap = MapViewOfFileEx( hFile, FILE_MAP_WRITE, 0, 0, 0, BASE_MEM );

   info.pBase = hMap;
   if (info.pBase == NULL) {
     return NULL;
   }

   /* Create a fixed sized heap.  Put the heap handle as well as the      */
   /* base heap address at the beginning of the shared memory.            */

   info.pHeap = _ucreate((char *)info.pBase + sizeof(info),
                 heap_size - sizeof(info),
                 !_BLOCK_CLEAN,
                 _HEAP_SHARED | _HEAP_REGULAR,
                 NULL, NULL);

    if (info.pBase == NULL) {
      return NULL;
    }

   memcpy(info.pBase, info, sizeof(info));

   if (_uopen(info.pHeap)) {            /* Open heap and check result    */
      return NULL;
   }

   return info.pHeap;

}
```

```
/*--------------------------------------------------------------------*/
/* termhp:                                                            */
/* Function to close and destroy the heap                             */
/*--------------------------------------------------------------------*/
static int termhp(Heap_t uheap)
{
   if (_uclose(uheap))                     /* close heap              */
      return 1;
   if (_udestroy(uheap, _FORCE))           /* force destruction of heap */
      return 1;

   UnmapViewOfFile(hMap);                  /* return memory to system   */
   CloseHandle(hFile);

   return 0;
}

/*--------------------------------------------------------------------*/
/* main:                                                              */
/* Main function to test creating, writing to and destroying a shared */
/* heap.                                                              */
/*--------------------------------------------------------------------*/
int main(void)
{
   int i, rc;                          /* Index and return code         */
   Heap_t uheap;                       /* heap to create                */
   void *init_block;                   /* initial block to use          */
   char *p;                            /* for allocating from heap      */

   /*                                                                   */
   /* call init function to create and open the heap                    */
   /*
   uheap = inithp(MEMORY_SIZE);
   if (uheap == NULL)                  /* check for success             */
      return 1;                        /* if failure, return non zero   */

   /*                                                                   */
   /* perform operations on uheap                                       */
   /*                                                                   */
   for (i = 1; i <= 5; i++)
   {
      p = _umalloc(uheap, 10);         /* allocate from uheap           */
      if (p == NULL)
         return 1;
      memset(p, 'M', _msize(p));       /* set all bytes in p to 'M'     */
      p = realloc(p,50);               /* reallocate from uheap         */
      if (p == NULL)
         return 1;
      memset(p, 'R', _msize(p));       /* set all bytes in p to 'R'     */
   }

   /*                                                                   */
   /* Start a second process which accesses the heap                    */
   /*                                                                   */
   if (system("memshr2.exe"))
     return 1;

   /*                                                                   */
   /* Take a look at the memory that we just wrote to. Note that memshr.c  */
   /* and memshr2.c should have been compiled specifying the            */
   /* alloc(debug[, yes]) flag.                                         */
   /*                                                                   */
   #ifdef DEBUG
     _udump_allocated(uheap, -1);
   #endif
```

```
    /*                                                                  */
    /* call term function to close and destroy the heap                 */
    /*                                                                  */
    rc = termhp(uheap);

    #ifdef DEBUG
      printf("memshr ending... rc = %d\n", rc);
    #endif

    return rc;
}
```

Example of a Shared User Heap - Child Process

```
/* Example of a Shared User Heap - Child Process shows
   the process started by the loop in the parent process.
   This process uses OpenFileMapping to access the shared memory
   by name, then extracts the heap handle for the heap created
   by the parent process. The process then opens the heap,
   makes it the default heap, and performs some operations
   on it in the loop. After the loop, the process replaces
   the old default heap, closes the user heap, and ends.


 */

 #include <umalloc.h>
 #include <stdio.h>
 #include <stdlib.h>
 #include <string.h>

 static HANDLE hFile;       /* Handle to memory file                        */
 static void*  hMap;        /* Handle to allocated memory                   */

 typedef struct mem_info {
    void  * pBase;
    Heap_t  pHeap;
 } MEM_INFO_T;

 /*--------------------------------------------------------------------------*/
 /* inithp:  Subprocess Version                                              */
 /* Function to create and open the heap with a named shared memory object */
 /*--------------------------------------------------------------------------*/
 static Heap_t inithp(void)
 {

    MEM_INFO_T info;                       /* Info structure               */

    /* Open the shared memory file by name.  The file is based on the      */
    /* system paging (swapper) file.                                        */

    hFile = OpenFileMapping(FILE_MAP_WRITE, FALSE, "MYNAME_SHAREMEM");

    if (hFile == NULL) {
      return NULL;
    }

    /* Figure out where to map this file by looking at the address in the   */
    /* shared memory where the memory was mapped in the parent process.     */

    hMap = MapViewOfFile( hFile, FILE_MAP_WRITE, 0, 0, sizeof(info) );

    if (hMap == NULL) {
      return NULL;
    }

    /* Extract the heap and base memory address from shared memory          */
```

```
   memcpy(info, hMap, sizeof(info));
   UnmapViewOfFile(hMap);

   hMap = MapViewOfFileEx( hFile, FILE_MAP_WRITE, 0, 0, 0, info.pBase );

   if (_uopen(info.pHeap)) {          /* Open heap and check result       */
      return NULL;
   }

   return info.pHeap;
}

/*--------------------------------------------------------------------------*/
/* termhp:                                                                  */
/* Function to close my view of the heap                                    */
/*--------------------------------------------------------------------------*/
static int termhp(Heap_t uheap)
{
   if (_uclose(uheap))                       /* close heap                */
      return 1;

   UnmapViewOfFile(hMap);                    /* return memory to system   */
   CloseHandle(hFile);

   return 0;
}

/*--------------------------------------------------------------------------*/
/* main:                                                                    */
/* Main function to test creating, writing to and destroying a shared     */
/* heap.                                                                    */
/*--------------------------------------------------------------------------*/
int main(void)
{
   int rc, i;                       /* for return code, loop iteration    */
   Heap_t uheap, oldheap;           /* heap to create, old default heap   */
   char *p;                         /* for allocating from the heap       */

   /*                                                                    */
   /* Get the heap storage from the shared memory                        */
   /*                                                                    */
   uheap = inithp();
   if (uheap == NULL)
     return 1;

   /*                                                                    */
   /* Register uheap as default runtime heap, save old default           */
   /*                                                                    */
   oldheap = _udefault(uheap);
   if (oldheap == NULL) {
      return termhp(uheap);
}

   /*                                                                    */
   /* Perform operations on uheap                                        */
   /*                                                                    */
   for (i = 1; i <= 5; i++)
   {
      p = malloc(10);     /* malloc uses default heap, which is now uheap*/
      memset(p, 'M', _msize(p));
   }

   /*                                                                    */
   /* Replace original default heap and check result                     */
   /*                                                                    */
   if (uheap != _udefault(oldheap)) {
```

```
       return termhp(uheap);
}

    /*                                                                       */
    /* Close my views of the heap                                            */
    /*                                                                       */
    rc = termhp(uheap);

    #ifdef DEBUG
      printf("Returning from memshr2 rc = %d\n", rc);
    #endif
    return rc;

 }
```

RELATED CONCEPTS

Memory Management Functions (page 25)
Managing Memory with Multiple Memory Heaps (page 27)
Types of Memory (page 29)
Debugging Memory Heaps (page 29)

RELATED TASKS

Create and Use a Fixed Size Heap (page 31)
Create and Use an Expandable Heap (page 33)
Debug Programs with Heap Memory (page 35)
Change the Default Heap Used in a Program (page 37)

Chapter 5. Program Optimization

Overview of Optimization

During optimization, the compiler changes unoptimized code sequences, derived from the source code, into equivalent optimized code sequences. The resulting code runs faster and usually takes less space. However, during optimization, compilation takes more time and space. In general, optimizing results in faster and smaller programs. In addition, when you optimize your code you may uncover bugs that were not evident before.

The decision to optimize for speed or for size depends on the goals for your application, and the nature of the application. When you choose options to enhance speed, the compiler generates the fastest instruction sequences possible, but these may not be the smallest possible. Similarly, when you choose options to reduce size, the compiler generates the smallest instruction sequences possible for the source code, but these may not be the fastest possible.

For larger programs which are not compute-intensive, optimizing for size might result in a faster program than one optimized for speed. This is because global effects such as improved paging and cache performance may outweigh the local effects of slower instruction sequences.

If both size and speed are important, consider balancing the performance by optimizing some modules for speed, and others for size. Determine which modules contain hotspots, and are compute-intensive: these should be optimized for speed. All other modules should be optimized for size. To find the right balance, you may need to experiment with different combinations of techniques.

When to Optimize
Optimize your code throughout your development cycle. Develop, test, and optimize incrementally rather than developing and testing and then optimizing the entire application at the end.

Because using optimization options transforms the code, the direct correspondence between source and object code is often lost. Therefore, debugging information is not accurate for programs compiled using the optimization option. Optimized code is also more sensitive to subtle coding errors. For these reasons, do not use the optimization options while you are developing your programs.

Optimization Levels in C or C++
The default is *not* to optimize your program. To optimize your program, specify the -qoptimization option.

.When you specify optimization, the compiler performs a complete control and data-flow analysis for each function. The compiler also uses global register allocation for the whole function, allowing many variables to be kept in registers rather than in memory.

RELATED CONCEPTS
Optimization Techniques Used by VisualAge C++ (page 46)
Enhanced Handling of Math and String Library Functions (page 48)
Debugging Optimized Code

© Copyright IBM Corp. 1998, 2002

RELATED TASKS

Find Faster I/O Techniques (page 48)
Optimize Your Application (page 49)
Reduce Function-Call Overhead (page 50)

RELATED REFERENCES

Coding Techniques That Can Improve Performance (page 51)
Optimization Options
Built-in Functions for PowerPC

Optimization Techniques Used by VisualAge C++

Technique	Description of Technique
Value Numbering	Involves constant propagation, expression elimination, and folding of several instructions into a single instruction.
Branch Optimizations	Rearranges the program code to minimize branching logic and to combine physically separate blocks of code.

Common Subexpression Elimination	In common expressions, the same value is recalculated in a subsequent expression. The duplicate expression can be eliminated by using the previous value. This step is done even for intermediate expressions within expressions. For example, if your program contains the following statements:
	`a = c + d;`
	`.`
	`.`
	`.`
	`f = c + d + e;`
	the common expression c + d is saved from its first evaluation and is used in the subsequent statement to determine the value of f.

Code Motion	If variables used in a computation within a loop are not altered within the loop, the calculation can be performed outside of the loop and the results used within the loop.

<table>
<tr>
<td>Invariant IF Code Floating (Unswitching)</td>
<td>

Removes invariant branching code from loops to allow opportunities for other optimizations.

For example, in the following code segment, the condition test and the conditional assignment:

```
if (a[i]>100.0) b[i]=a[i]-3.7;
    x+=a[j]+b[i];
```

do not change during execution of the inner loop.

```
for (i=0;i<1000;i++) {
   for (j=0;j<1000;j++) {
      if (a[i]>100.0) b[i]=a[i]-3.7;
      x+=a[j]+b[i];
   }
}
```

The compiler translates the code into a machine-language loop that executes as:

```
for (i=0;i<1000;i++) {
   if (a[i]>100.00) {
       b[i]=a[i]-3.7;
       for (j=0;j<1000;j++) {
          x+=a[j]+b[i];
      }
   }
   else {
      for (j=0;j<1000;j++) {
         x+=a[j]+b[i];
      }
   }
}
```

</td>
</tr>
<tr>
<td>Reassociation</td>
<td>Rearranges the sequence of calculations in an array-subscript expression, producing more candidates for common-expression elimination.</td>
</tr>
<tr>
<td>Strength Reduction</td>
<td>Replaces less efficient instructions with more efficient ones. For example, in array subscripting, an add instruction replaces a multiply instruction.</td>
</tr>
<tr>
<td>Constant Propagation</td>
<td>Constants used in an expression are combined, and new ones are generated. Some implicit conversions between integer and floating-point types are done.</td>
</tr>
<tr>
<td>Store Motion</td>
<td>Moves store instructions out of loops.</td>
</tr>
<tr>
<td>Dead Store Elimination</td>
<td>Eliminates stores when the value stored is never referred to again. For example, if two stores to the same location have no intervening load, the first store is unnecessary and is removed.</td>
</tr>
<tr>
<td>Dead Code Elimination</td>
<td>Eliminates code that cannot be reached or code whose results are not subsequently used.</td>
</tr>
<tr>
<td>Inlining</td>
<td>Replaces function calls with actual program code.</td>
</tr>
</table>

Instruction Scheduling	Reorders instructions to minimize execution time.
Global Register Allocation	Allocates variables and expressions to available hardware registers using a *graph coloring* algorithm.

RELATED CONCEPTS

Enhanced Handling of Math and String Library Functions (page 48)

Enhanced Handling of Math and String Library Functions

VisualAge C++ enhances run-time performance for C and C++ applications by substituting tuned machine code for calls to the most commonly used string processing and numerical functions in the C and C++ Standard Libraries. When optimization is enabled, functions declared in <string.h> and <math.h> are substituted when either <math.h> or <string.h> is included in a source file.

You can prevent this substitution without disabling optimization by undefining the __MATH__ and/or __STR__ preprocessor macros in your configuration file. Run-time performance of your application may be affected.

RELATED CONCEPTS

Optimization Techniques Used by VisualAge C++ (page 46)

RELATED TASKS

Optimize String Manipulation (page 55)

RELATED REFERENCES

-U Compiler Option

Find Faster I/O Techniques

There are a number of ways to improve your program's performance of input and output:

- Use binary streams instead of text streams. In binary streams, data is not changed on input or output.
- Use the low-level I/O functions, such as open and close. These functions are faster and more specific to the application than the stream I/O functions like fopen and fclose. You must provide your own buffering for the low-level functions.
- If you do your own I/O buffering, make the buffer a multiple of 4K, which is the size of a page. Because malloc uses extra storage as overhead, allocating storage in a multiple of the page size actually results in more pages being allocated than required.
- If you know you have to process an entire file, determine the size of the data to be read in, allocate a single buffer to read it to, read the whole file into that buffer at once using Read, and then process the data in the buffer. This reduces disk I/O, provided the file is not so big that excessive swapping will occur. Consider using the mmap function to access the file.

- Instead of scanf and fscanf, use fgets to read in a string, and then use one of atoi, atol, atof, or _atold to convert it to the appropriate format.
- Use sprintf only for complex formatting. For simpler formatting, such as string concatenation, use a more specific string function.
- When reading input, read in a whole line at once rather than one character at a time.

RELATED CONCEPTS

Overview of Optimization (page 45)

Optimize Your Application

Because the size of your application affects both the load time and the run-time characteristics, it is best to do size tuning before performance tuning. Changes you should already have considered include:

- choosing efficient algorithms with small memory footprints
- avoiding duplicate copies of data

These are guidelines only. Remember that the results of optimization depend to a great extent on the code being optimized.

Use Function Arguments

Optimization is effective when function arguments are used. It is usually better to pass a value as an argument to a function than to let the function take the value from a global variable.

The #pragma isolated_call preprocessor directive lists functions that have no side effects and do not depend on side effects. Using the pragma to list functions that do not have side effects, that is, that do not modify global storage, can improve the run-time performance of optimized code.

Declare Nonmember Functions as Static

Declaring nonmember functions as static whenever possible will speed up calls to the function.

Use Multiplication Rather Than Division

Wherever possible, use multiplication rather than division. For example:

```
x*(1.0/3.0);
```

produces faster code than:

```
x/3.0;
```

Assigning the reciprocal of the divisor to a temporary variable and then multiplying by that variable is beneficial, especially if you divide many values by the same number in your code. The compiler attempts to do this when -qnostrict is specified.

Expand Loops

If your program contains a short, heavily referenced for loop, consider expanding the code to a straight sequence of statements. For example:

```
array[0] = b[k+1]*c[m+1];
array[1] = b[k+2]*c[m+2];
array[2] = b[k+3]*c[m+3];
array[3] = b[k+4]*c[m+4];
array[4] = b[k+5]*c[m+5];
```

will run faster than:

```
for (i = 0; i < 5; i++)
   array[i] = b[k+i]*c[m+i];
```

The compiler performs automatic unrolling of small inner loops when -qopt=2 is specified. In this example, the compiler unrolls the loop fully.

Minimize the Size of Object Files

To minimize the size of object files, specify the -qcompact compiler option. Using this option may increase execution time.

RELATED CONCEPTS

Overview of Optimization (page 45)
Debugging Optimized Code

RELATED REFERENCES

Coding Techniques That Can Improve Performance (page 51)
-qoptimize Compiler Option

Reduce Function-Call Overhead

Whether you are writing a function or calling a library function, there are a few things you should keep in mind:

- C++ Use virtual functions only when necessary. They are usually compiled to be indirect calls, which are slower than direct calls.
- C++ Usually, you should not declare virtual functions inline. If all virtual functions in a class are inline, the virtual function table and all the virtual function bodies will be replicated in each compilation unit that uses the class.
- Unless absolutely necessary, do not use function pointers; call functions directly.
- C Fully prototype all functions. A full prototype gives the compiler and optimizer complete information about the types of the parameters. As a result, promotions from unwidened types to widened types are not required, and parameters may be passed in appropriate registers.
- C Avoid using unprototyped variable argument functions.
- When designing a function, place the most used parameters in one of the left-most positions in the function prototype. The left-most 8 words of parameters will be passed in registers.
- Avoid passing structures or unions as function parameters or returning a structure or a union. Passing such aggregates requires the compiler to copy and store many values. This is worse in C++ programs in which class objects are passed by value, a constructor and destructor are called when the function is called. Instead, pass or return a pointer to the structure or union, or pass it by reference.
- Pass atomic types (like int and short) by value rather than passing by reference, wherever possible.

- If your function exits by returning the value of another function with the same parameters that were passed to your function, put the parameters in the same order in the function prototypes. The compiler can then branch directly to the other function.
- Use the intrinsic and built-in functions, which include string manipulation, floating-point, and trigonometric functions, instead of coding your own. Intrinsic functions require less overhead and are faster than a function call, and often allow the compiler to perform better optimization. Your functions are automatically mapped to intrinsic functions if you include the VisualAge C++ header files.
  C This mapping is overridden if you #undef the macro name.
- Use recursion only where necessary. Because recursion involves building a stack frame, an iterative solution is usually faster than a recursive one, except when the function exits by calling itself.

RELATED CONCEPTS
Overview of Optimization (page 45)

RELATED TASKS
Optimize Your Application (page 49)

Coding Techniques That Can Improve Performance

Because the size of your application affects both the load time and the run-time characteristics, it is best to do size tuning before performance tuning. Changes you should already have considered include:

- choosing efficient algorithms with small memory footprints
- avoiding duplicate copies of data
- structuring data to minimize padding between items

If you have not already checked your program for these types of improvements, it is best to do so before trying any of the techniques below. These are guidelines only. Remember that the results of optimization depend to a great extent on the code being optimized.

- Minimize the use of external (extern) variables to improve aliasing information and TOC loads.
- Use the const qualifier whenever possible.
- When declaring C++ functions, use the const specifier whenever possible.
- Use static functions whenever possible.
- Avoid taking the address of local variables. If you use a local variable as a temporary variable and must take its address, avoid reusing the temporary variable. Taking the address of a local variable inhibits optimizations that would otherwise be done on calculations involving that variable.
- Avoid using long long int types, except where absolutely necessary. Extra instructions must be generated to perform operations on such data types.
- Use unsigned types whenever possible. Faster code can be generated for division or modulo operations involving unsigned types.
- Make sure your data is aligned on a multiple of its size. For example, align double types on an 8-byte boundary.

- Use constants instead of variables where possible. The optimizer will be able to do a better job reducing run-time calculations by doing them at compile-time instead. For instance, if a loop body has a constant number of iterations, use constants in the loop condition to improve optimization; (`for (i=0; i<4; i++)`can be better optimized than `for (i=0; i<x; i++)`).
- Avoid goto statements that jump into the middle of loops. Such statements inhibit certain optimizations.
- Where possible, the most frequently accessed member of a structure should be placed first within the structure. Since no offset is needed to access the first member, doing so can improve size and speed.
- Improve the predictability of your code by making the fall-through path more probable. That is, code like `if (error) {handle error} else {real code}`should be written as `if (!error) {real code} else {error}`.
- If one or two cases of a switch are typically executed much more frequently than other cases, break out those cases by handling them separately before the switch statement.
- Inline your functions selectively. Inlined functions require less overhead and are generally faster than a function call. The best candidates for inlining are small functions that are called frequently from a few places. Large functions and functions that are called rarely may not be good candidates for inlining. Be sure to inline all functions that just load or store a value.
- C++ Use try blocks for exception handling only when necessary because they can inhibit optimization.
- C++ Avoid performing a *deep copy* if a *shallow copy* is all you require. For an object that contains pointers to other objects, a shallow copy copies only the pointers and not the objects to which they point. The result is two objects that point to the same contained object. A deep copy, however, copies the pointers and the objects they point to, as well as any pointers or objects contained within that object, and so on.
- C++ When you use the Collection classes from the IBM Open Class Library to create classes, use a high level of abstraction. After you establish the type of access to your class, you can create more specific implementations. This can improve performance with minimal code change.
- Use constant arguments in inlined functions whenever possible. Functions with constant arguments provide more opportunities for optimization.
- If you have a function that is called many times from a few functions, but infrequently from others, create a copy of the function with a different name and inline it only in the functions that call it often.
- Pass small const parameters by value; pass large parameters by reference.

RELATED CONCEPTS

Overview of Optimization (page 45)

RELATED TASKS

Reduce Function-Call Overhead (page 50)
Find Faster I/O Techniques (page 48)

Memory Management and Performance

Because C++ objects are often allocated from the heap and have limited scope, memory use in C++ programs affects performance more than in C programs. Points below that apply specifically to C++ are identified by ▶ C++

- When you declare or define structures or C++ classes, take into account the alignment of data types. Declare the largest members first to reduce wasted space between members and to reduce the number of boundaries the compiler must cross. The alignment is especially important if you pack your structure or class.
- ▶ C++ Tailor your own new and delete operators, using the system new to allocate an array of objects for a class, and using that class's new to allocate individual objects from the array.
- ▶ C++ Ensure that objects that are no longer needed are freed or otherwise made available for reuse. One way to do this is to use an *object manager*. Each time you create an instance of an object, pass the pointer to that object to the object manager. The object manager maintains a list of these pointers. To access an object, you can call an object manager member function to return the information to you. The object manager can then manage memory usage and object reuse.
- ▶ C++ Avoid copying large, complex objects.

RELATED REFERENCES

Coding Techniques That Can Improve Performance (page 51)
Memory Management Functions (page 25)

Mixed-Mode Arithmetic

Avoid forcing the compiler to convert numbers between integer and floating-point internal representations. Conversions require several instructions, including some double-precision floating-point arithmetic, and most importantly, a store and a load, and on some CPUs, a large memory/cache delay between storing and loading. For example:

```
float array[10];
float x = 1.0;
int i;
for (i = 0; i< 9; i++)  {     /* No conversions needed */
     array[i] = array[i]*x;
      x = x + 1.0;
}
for (i = 0; i< 9; i++)        /* Multiple conversions needed */
   array[i] = array[i]*i;
```

When you must use mixed-mode arithmetic, code the integer and floating-point arithmetic in separate computations wherever possible.

RELATED CONCEPTS

Overview of Optimization (page 45)

Expressions

If components of an expression are duplicate expressions, code them either at the left end of the expression or within parentheses. For example:

```
a = b*(x*y*z);       /* Duplicates recognized */
c = x*y*z*d;
e = f + (x + y);
```

```
g = x + y + h;
a = b*x*y*z;        /* No duplicates recognized */
c = x*y*z*d;
e = f + x + y;
g = x + y + h;
```

When components of an expression in a loop are constant, or loop-invariant, code the expressions either at the left end of the expression, or within parentheses. If c, d, and e are constant and v, w, and x are variable, the following examples show the difference in evaluation:

```
v*w*x*(c*d*e);          /* Constant expressions recognized   */
c + d + e + v + w + x;
v*w*x*c*d*e;            /* Optimization required for constant */
v + w + x + c + d + e;  /* expressions to be recognized      */
```

When -qoptimize=2 is used, integer constant or loop-invariant expressions will be recognized regardless of order or parenthesizing.

RELATED CONCEPTS

Overview of Optimization (page 45)

RELATED REFERENCES

Optimization Options

Variables and Optimization

Use local variables, preferably automatic variables, as much as possible. The compiler can accurately analyze the use of local variables, but it has to make several worst-case assumptions about global variables. These assumptions tend to hinder optimization. For example, if you write a function that uses external variables, and that function also calls external functions, the compiler assumes that every call to an external function could change the value of every external variable. If you know that none of the function calls affects the global variables that you are using, and you have to read them frequently with function calls interspersed, copy the global variables to local variables and then use these local variables. The compiler can then perform optimization that it could not otherwise perform.

If you must use global variables, use static variables with file scope rather than external variables wherever possible. In a file with several related functions and static variables, the optimizer can gather and use more information about how the variables are affected.

To access an external variable, the compiler has to make an extra memory access to obtain the address of the variable. When the compiler removes extraneous address loads, it has to use a register to keep the address. Using many external variables simultaneously takes up many registers. Those that cannot fit into registers during optimization are spilled into memory. Because all elements of an external structure use the same base address, you should group external data into structures or arrays wherever it makes sense to do so.

The #pragma isolated_call preprocessor directive can improve the run-time performance of optimized code by allowing the compiler to make less pessimistic assumptions about the storage of external and static variables. Isolated_call functions with constant or loop-invariant parameters may be moved out of loops, and multiple calls with the same parameters may be replaced with a single call.

Because the compiler treats register variables the same as it does automatic variables, you do not gain anything by declaring register variables. Note that this differs from other implementations, where using the register attribute can greatly affect program performance.

RELATED CONCEPTS

Overview of Optimization (page 45)

RELATED REFERENCES

Example of Using Volatile Variables (page 22)

Optimize String Manipulation

The handling of string operations can affect the performance of your program.

- When you store strings into allocated storage, align the start of the string on a 4-byte boundary. This allows the best performance of the string functions. The compiler performs this alignment for all strings it allocates.
- Keep track of the length of your strings. If you know the length of your string, you can use mem functions instead of str functions. For example, memcpy is faster than strcpy because it does not have to search for the end of the string.
- When manipulating strings using mem functions, faster code will be generated if the *count* parameter is a constant rather than a variable. This is especially true for small count values.
- Avoid using strtok. Because this function is very general, you can probably write a function more specific to your application and get better performance.

String literals are read-only by default. Placing them into read-only memory allows for certain types of optimizations and also causes the compiler to put out only one copy of strings that are used in more than one place. If you use the intrinsic string functions, the compiler can better optimize them if it knows that any string literals it is operating on will not be changed.

You can explicitly set strings to read-only by using #pragma strings (readonly) in your source files or -qro to avoid changing your source files.

RELATED CONCEPTS

Overview of Optimization (page 45)

RELATED REFERENCES

Enhanced Handling of Math and String Library Functions (page 48)

Chapter 6. Floating Point Operations

Floating Point Hardware

Single precision values have an approximate range of 10(-38) to 10(+38), with about 7 decimal digits of precision. Double precision values have an approximate range of 10(-308) to 10(+308) and precision of about 16 decimal digits. Quadruple precision values have the same range as double precision values, but the precision is about 29 decimal points.

When results must be converted to single precision, rounding operations are used. A rounding operation produces the correct single-precision value based on the IEEE rounding mode in effect. Because explicit rounding operations are required, single-precision computations are often slower than double precision computations. On many other machines the reverse is true: single-precision operations are faster than double-precision operations. Code ported from other systems can show different performance on a RISC System/6000 Power or Power2 computer.

The floating-point hardware in the RISC/6000 Power and Power2 families performs all computations in IEEE double precision, equivalent to double in C and C++ programs. Single-precision (float) values are automatically converted to double precision before they are used, and all results are calculated in double precision. Double precision provides greater range and precision than single precision does. Quadruple precision (long double) operations are slower than double precision operations, and some are significantly slower.

The RISC System/6000 hardware also provides a special set of double-precision operations that multiply two numbers and add a third number to the product. These combined multiply-add (maf) operations are performed in the same time as a multiply or an add operation alone. The maf functions provide an extension to the IEEE standard because they perform the multiply and add with one (rather than two) rounding errors. The maf functions are both faster and more accurate than the equivalent separate operations. Use the gen(float,maf,no) option to suppress the generation of these multiply-add instructions for greater compatibility with the accuracy available on other systems.

Note: The PowerPC hardware platforms can perform most computations in IEEE single precision. The instruction set used to generate code is determined by the setting of the -qarch option.

Detecting Floating-Point Exceptions
A number of floating-point exceptions can be detected by the floating-point hardware: invalid operation, division by zero, overflow, underflow, and inexact. By default, all exceptions are ignored. However, if you use the -qflttrap option, any or all of these exceptions can be detected. In addition, when you add suitable support code to your program, program execution can continue after an exception occurs, and you can then modify the results of operations causing exceptions.

Refer to “Floating-Point Processor Overview” and “Floating-Point Exceptions” in the *AIX Assembler Language Reference* for more information about RISC System/6000 and pSeries floating-point processing.

© Copyright IBM Corp. 1998, 2002

RELATED CONCEPTS
Compile-Time Floating Point Arithmetic (page 58)

RELATED REFERENCES
Rounding Mode Restrictions (page 59)
-qfloat Compiler Option

Compile-Time Floating-Point Arithmetic

The compiler attempts to perform as much floating-point arithmetic as possible at compile time. Floating-point operations with constant operands are folded, replacing the operation with the result calculated at compile time. When optimization is enabled, more folding might occur than when optimization is not enabled.

All compile-time folding of floating-point computations can be suppressed using the -qfloat=nofold option. Alternatively, the IEEE rounding mode used in compile-time arithmetic can be controlled using the -qfloat options.

Compile-time floating-point arithmetic can have two effects on program results:

- In specific cases, the result of a computation at compile time might differ slightly from the result that would have been calculated at run time. The reason is that more rounding operations occur at compile time. For example, where a maf operation might be used at run time, separate multiply and add operations might be used at compile time, producing a slightly different result.
- Computations that produce exceptions can be folded to the IEEE result that would have been produced by default in a run-time operation. This would prevent an exception from occurring at run time. When using the flttrap option, you should consider using the gen(float,fold,no) option.

In general, code that affects the rounding mode at run time should be compiled with the -y option that matches the rounding mode intended at run time. For example, when the following program:

```
main ()
{
  union uu
  {
    float x;
    int i;
  } u;
  volatile float one, three;
  u.x=1.0/3.0;
  printf("1/3=%8x \n", u.i);
  one=1.0
  three=3.0;
  u.x=one/three;
  printf ("1/3=%8x \n", u.i);
}
```

is compiled using -yz, the expression 1.0/3.0 is folded by the compiler at compile time into a double-precision result. This result is then converted to single-precision and then stored in float u.x. The -qfloat=nofold option can be specified to suppress all compile-time folding of floating-point computations. The -y option only affects compile-time rounding of floating-point computations, but does *not* affect run time rounding. The code fragment:

```
one = 1.0;
three = 3.0;
x = one/three;
```

is evaluated at run time in single-precision. Here, the default run-time rounding of "round to nearest" is still in effect and takes precedence over the compile-time specification of "round to zero". The output of this program is:

```
1/3=3eaaaaaa
1/3=3eaaaaab
```

RELATED CONCEPTS

Rounding Mode Restrictions (page 59)

RELATED REFERENCES

-qfloat Compiler Option

Rounding Mode Restrictions

The floating-point rounding mode can only be changed at the beginning and end of a function. It cannot be changed across a function call, and if it is changed within a function, it must be restored before returning to the calling routine.

RELATED CONCEPTS

Compile-Time Floating-Point Arithmetic (page 58)

Chapter 7. USL Input/Output Stream Classes

USL I/O Streaming

This section refers to the USL I/O Stream Library.

VisualAge C++ comes with ANSI-compliant stream classes. We recommend that you use these stream classes instead to develop thread-safe applications. The ANSI-compliant stream classes are part of the Standard C++ Library.

The USL I/O Stream Library provides 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 USL I/O Stream Library:

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.
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 USL I/O Stream Library predefines streams for standard input, standard output, and standard error. 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 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.

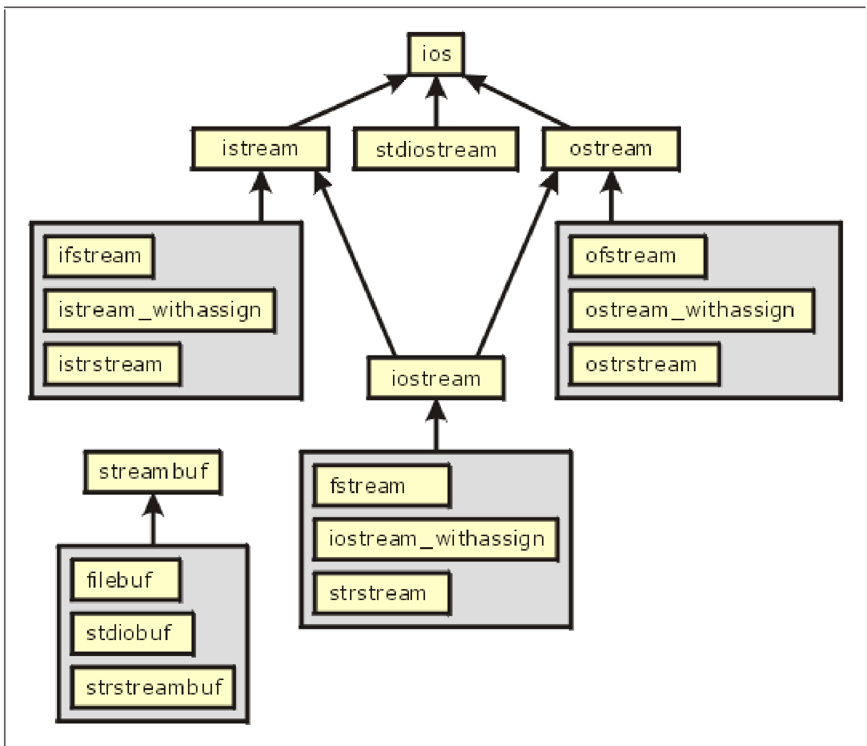
Input and Output for User-Defined Classes
You can overload the input and output operators for the classes that you create yourself. Once you have overloaded the input and output operators for a class, you can perform input and output operations on objects of that class in the same way that you would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

RELATED CONCEPTS
USL I/O Stream Class Hierarchy (page 62)
USL I/O Stream Header Files (page 63)
Stream Buffers (page 69)
Format State Flags (page 71)
Manipulators (page 71)
The I/O Stream Classes and stdio.h

© Copyright IBM Corp. 1998, 2002

The USL I/O Stream Class Hierarchy

The USL I/O Stream Library has two base classes, streambuf and ios:



The streambuf class implements *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.

The istream class 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

The ostream class is the output stream class. It implements stream buffer output, or output operations. The following classes are derived from ostream:
- ostrstream

- ofstream
- ostream_withassign
- iostream

The iostream class combines istream and ostream to implement input and output to stream buffers. The following classes are derived from iostream:
- strstream
- iostream_withassign
- fstream

The USL I/O Stream Library also defines other classes, including fstreambase and strstreambase. These classes are meant for the internal use of the USL I/O Stream Library. Do not use them directly.

RELATED CONCEPTS
USL I/O Streaming (page 61)

USL I/O Stream Header Files

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

The header file iostream.h contains declarations for the basic classes:
- strstreambuf
- ios
- istream
- istream_withassign
- ostream
- ostream_withassign
- iostream
- iostream_withassign

The header file fstream.h contains declarations for the classes that deal with files:
- filebuf
- ifstream
- ofstream
- fstream

The header file stdiostream.h contains declarations for stdiobuf and stdiostream, the classes that specialize streambuf and ios, respectively, to use the FILE structures defined in the C header file stdio.h.
400 The 8.3 file naming convention compliant name of this file is stdiostr.h.

The header file strstream.h contains declarations for the classes that deal with character strings.
400 The 8.3 file naming convention compliant name of this file is strstrea.h.
The first "str" in each of these names stands for "string":
- istrstream
- ostrstream
- strstream

- strstreambuf

The header file 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.

The header file stream.h is used for compatibility with earlier versions of the USL I/O Stream Library. It includes iostream.h, fstream.h, stdiostream.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.

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

RELATED CONCEPTS

USL I/O Streaming (page 61)

Open a File for Input and Read from the File

Use the following steps to open a file for input and to read from the file.

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.
   z/OS z/OS C/C++ provides overloads of the fstream and ifstream constructors and their open() functions, which allow you to specify file attributes such as lrecl and recfm.
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.

Construct an fstream or ifstream Object for Input

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

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

```
#include <fstream.h>
int main(int argc, char *argv[]) {
   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>
int main(int argc, char *argv[]) {
   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>
int main(int argc, char *argv[]) {
   char q[40];
   fstream myfile("test.txt",ios::in); // open the file for input
   myfile >> q;                       // input from myfile into q
   myfile.close();                    // close the file
   myfile.open("test.txt",ios::app);   // reopen the file for output
   myfile << q << endl;               // output from q to myfile
   myfile.close();                    // close the file
   return 0;
}
```

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. Let's assume that the contents of the file test.txt before the program is run are:

```
barbers often shave
```

In this case, 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.txt",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.txt would be appended to test.out instead of to test.txt itself.

Read Input from a File

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

stdin.C	filein.C

`#include <iostream.h>`
 	
`int main(int argc, char *argv[]) {`
`   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;`
`   return 0;`
`   }`
	`#include <fstream.h>`
	
	`int main(int argc, char *argv[]) {`
	`   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;`
	`   return 0;`
	`   }`

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.

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.

All techniques for reading input from the standard input stream can also 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.

RELATED CONCEPTS

USL I/O Streaming (page 61)

RELATED TASKS

Combine Input and Output of Different Types
Use Predefined Streams
Receive Input from Standard Input
Display Output from Standard Output or Standard Error
Flush Output Streams with endl and flush
Parse Multiple Inputs
Open a File for Output and Write to the File (page 67)
Associate a File with a Standard Input or Output Stream
Move through a File with filebuf Functions
Define an Input Operator for a Class Type
Define an Output Operator for a Class Type
Correct Input Stream Errors
Format Stream Output
Define Your Own Format State Flags
Manipulate Strings with the strstream Classes (page 68)
Create Manipulators (page 73)

Open a File for Output and Write to the File

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>
int main(int argc, char *argv[]) {
   fstream file1("file1.out",ios::app);
   ofstream file2("file2.out");
   ofstream file3;
   file3.open("file3.out");
   return 0;
}
```

   You must specify one or more open modes when you open the file, unless you declare the object as an ofstream object. 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.

   z/OS z/OS C/C++ provides overloads of the fstream and ofstream constructors and their open() functions, which allow you to specify file attributes such as lrecl and recfm. Refer to the IBM Open Class Reference for more information.
2. Use the output operator or ostream member functions to perform output to the file.
3. Close the file using the close() member function of fstream.

When you define an output operator for a class type, this output operator is available both to the predefined output streams and to any output streams you define.

RELATED CONCEPTS

USL I/O Streaming (page 61)

RELATED TASKS

Combine Input and Output of Different Types
Use Predefined Streams
Receive Input from Standard Input
Display Output from Standard Output or Standard Error
Flush Output Streams with endl and flush
Parse Multiple Inputs
Open a File for Input and Read from the File (page 64)
Associate a File with a Standard Input or Output Stream
Move through a File with filebuf Functions
Define an Input Operator for a Class Type
Define an Output Operator for a Class Type
Correct Input Stream Errors
Format Stream Output
Define Your Own Format State Flags
Manipulate Strings with the strstream Classes (page 68)
Create Manipulators (page 73)

Manipulate Strings with the strstream Classes

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.

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.
- 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*);
int main(int argc, char *argv[])
{
   cout << add("1 27 32.12 518") << endl;
   return 0;
}
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.
```

RELATED CONCEPTS

USL I/O Streaming (page 61)

RELATED TASKS

Combine Input and Output of Different Types
Use Predefined Streams
Receive Input from Standard Input
Display Output from Standard Output or Standard Error
Flush Output Streams with endl and flush
Parse Multiple Inputs
Open and Read from Files (page 64)
Open and Write to Files (page 67)
Associate a File with a Standard Input or Output Stream
Move through a File with filebuf Functions
Define an Input Operator for a Class Type
Define an Output Operator for a Class Type
Correct Input Stream Errors
Format Stream Output
Define Your Own Format State Flags
Create Manipulators (page 73)

Stream Buffers

One of the most important concepts in the USL I/O Stream Library 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.

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.

z/OS The main reason to use stream buffers on the z/OS is to ensure optimal portability.

How is a stream buffer implemented?

A stream buffer is implemented as an array of bytes. For each stream buffer, pointers are defined that point to elements in this array to define the *get area* (the

space that is available to accept bytes from the ultimate producer), and the *put area* (the space that is available to store bytes that are on their way to the ultimate consumer.

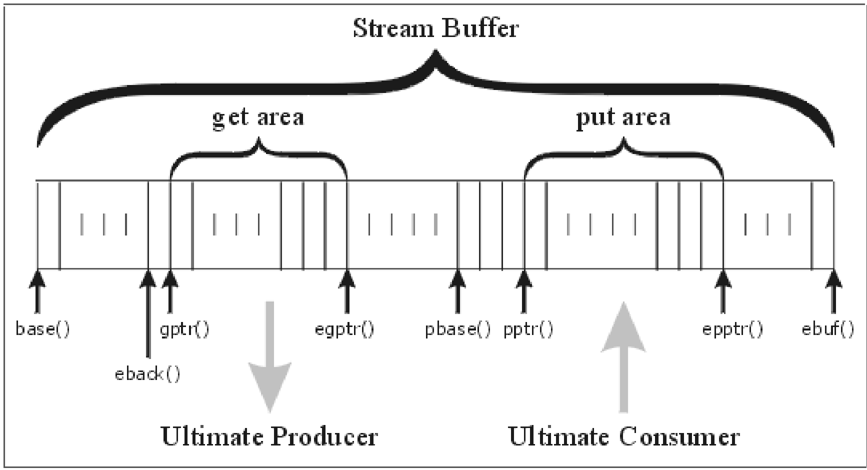
A stream buffer does not necessarily have separate get and put areas:
- A stream buffer that is used for input, such as one that is attached to an istream object, has a get area.
- A stream buffer that is used for output, such as the one that is attached to an ostream object, has a put area.
- A stream buffer that is used for both input and output, such as the 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:

Member function	Description
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.
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.

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.

The following diagram is the structure of a stream buffer:



RELATED CONCEPTS

USL I/O Streaming (page 61)

Format State Flags

The ios class defines an enumeration of format state flags that you can use to affect the formatting of data in USL 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

RELATED CONCEPTS

USL I/O Streaming (page 61)

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.

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* and *parameterized*.

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.

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. With manipulators you can place the change to a stream's state within a list of outputs for that stream.

RELATED CONCEPTS
USL I/O Streaming (page 61)

RELATED TASKS
Create Manipulators (page 73)

Thread Safety and USL I/O Streaming

The USL I/O Stream Library provides thread safety at the object level. This means that it is safe to have multiple threads manipulate the same object. This library provides streaming operators for the built in C++ types. With object level thread safety, the output from one streaming operator will be streamed in entirety before the next. However, in a multi-threaded environment, there is no guarantee that the output from one streaming operator on the same thread will appear immediately after the output from the preceding streaming operator. For example, given the following scenario, either result may occur:

Scenario:

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

Result:

Desired	**`anInt1 aString1 anInt2 aString2`**
Possible	`anInt1 aString1 aString2 anInt2`

If order of output from separate threads is important, then explicit programmer serialization is required.

z/OS On z/OS, to run in a multi-threaded environment, the z/OS UNIX kernel must be available and active.

RELATED CONCEPTS
USL I/O Streaming (page 61)

Create Manipulators

Create Simple Manipulators for Your Own Types
The USL 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!
----------------------------------------------------------------
```

Create Parameterized Manipulators for Your Own Types
The USL 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 an APP Parameterized manipulator, choose a class that has APP in its name (an APP class, also known as an *applicator*). If you want to define a MANIP Parameterized manipulator, 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. Define the manipulator.

   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.

RELATED CONCEPTS

USL I/O Streaming (page 61)

RELATED TASKS

Define an APP Parameterized Manipulator (page 74)
Define a MANIP Parameterized Manipulator (page 75)
Define Nonassociative Parameterized Manipulators (page 76)

Define 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;
int main(int argc, char *argv[]) {
   my_class obj("Hello",'-',10);
   cout << pre_print(obj) << endl;
   return 0;
}
```

This program produces the following output:

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

RELATED CONCEPTS

USL I/O Streaming (page 61)

RELATED TASKS

Create Manipulators (page 73)
Define a MANIP Parameterized Manipulator (page 75)
Define Nonassociative Parameterized Manipulators (page 76)

Define 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);
}
int main(int argc, char *argv[]) {
   my_class obj("Hello",'-',10);
   cout << pre_print(obj) << "\0" << endl;
   return 0;
}
```

This example produces the following output:

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

RELATED CONCEPTS

USL I/O Streaming (page 61)

RELATED TASKS

Create Manipulators (page 73)
Define an APP Parameterized Manipulator (page 74)
Define Nonassociative Parameterized Manipulators (page 76)

Define 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;

int main(int argc, char *argv[]) {
   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;
   return 0;
}
```

RELATED CONCEPTS

USL I/O Streaming (page 61)

RELATED TASKS

Create Manipulators (page 73)
Define an APP Parameterized Manipulator (page 74)
Define a MANIP Parameterized Manipulator (page 75)

Chapter 8. USL Complex Math Classes

Complex Mathematics Library Overview

The Complex Mathematics Library provides you with the facilities to manipulate complex numbers and to perform standard mathematical operations on them. This library is comprised of two classes:

- complex is the class that lets you manipulate complex numbers
- c_exception is the class that you use to handle errors created by the functions and operations in the complex class.

The Complex Mathematics Library provides you with the following functionality:

- Mathematical operators with the same precedence as the corresponding real operators. With these operators, you can code expressions on complex numbers.
- Mathematical, trigonometric, magnitude, and conversion functions as friend functions of complex objects.
- Predefined mathematical constants.
- Input and output operators for USL I/O Stream Library input and output: 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 c_exception class to handle errors. You can also define your own version of the error handling function.

RELATED CONCEPTS

Review of Complex Numbers (page 77)
Header Files and Constants for the complex and c_exception Classes (page 78)
Mathematical Operators for complex (page 79)
Friend Functions for complex (page 80)
Input and Output Operators for complex (page 81)
Error Functions (page 81)

RELATED TASKS

Construct complex Objects (page 82)
Handle complex Mathematics Errors (page 86)
Example: Calculate Roots (page 88)
Example: Use Equality and Inequality Operators (page 90)

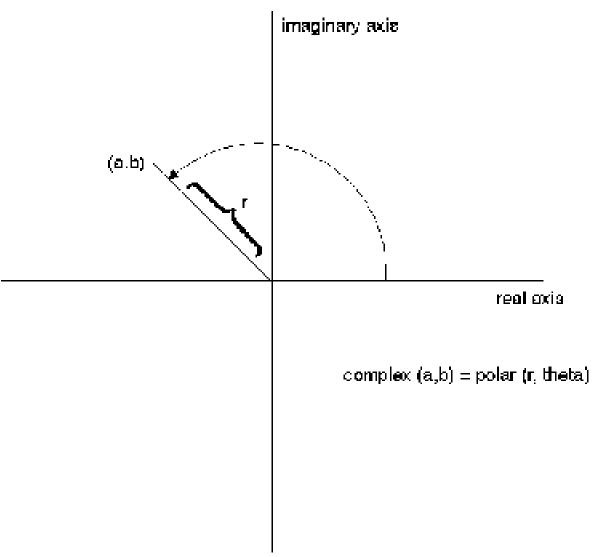
Review of Complex Numbers

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:

- $(a, b) + (c, d) = (a + c, b + d)$
- $(a, b) - (c, d) = (a - c, b - d)$
- $(a, b) * (c, d) = (ac - bd, ad + bc)$
- $(a, b) / (c, d) = ((ac + bd) / (c^2 + d^2), (bc - ad) / (c^2 + d^2))$

© Copyright IBM Corp. 1998, 2002

- 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. The following figure illustrates the polar representation (*r*,*theta*) of the complex number (*a*, *b*).



RELATED CONCEPTS

Complex Mathematics Library Overview (page 77)
Header Files and Constants for the complex and c_exception Classes (page 78)
Mathematical Operators for complex (page 79)
Friend Functions for complex (page 80)
Input and Output Operators for complex (page 81)
Error Functions (page 81)

RELATED TASKS

Construct complex Objects (page 82)
Example: Calculate Roots (page 88)
Example: Use Equality and Inequality Operators (page 90)

Header Files and Constants for the complex and c_exception Classes

To use the complex or c_exception classes, you must include the following statement in any file using these classes:

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

Constants Defined in complex.h

The following table lists the mathematical constants that the Complex Mathematics Library defines.

Constant Name	Description
M_E	The constant *e*

Constant Name	Description
M_LOG2E	The logarithm of *e* to the base 2
M_LOG10E	The logarithm of *e* to the base 10
M_LN2	The natural logarithm of 2
M_LN10	The natural logarithm of 10
M_PI	π (pi)
M_PI_2	π (pi) divided by two
M_PI_4	π (pi) divided by four
M_1_PI	1/ π (1/pi)
M_2_PI	2/ π (2/pi)
M_2_SQRTPI	2 divided by the square root of π (pi)
M_SQRT2	The square root of 2
M_SQRT1_2	The square root of 1/2

RELATED CONCEPTS

Complex Mathematics Library Overview (page 77)
Review of Complex Numbers (page 77)
Mathematical Operators for complex (page 79)
Friend Functions for complex (page 80)
Input and Output Operators for complex (page 81)
Error Functions (page 81)

RELATED TASKS

Construct complex Objects (page 82)

Mathematical Operators for complex

The complex class defines a set of mathematical operators with the same precedence as the corresponding real operators. With the following operators, you can code expressions on complex numbers:

- operator + (addition)
- operator * (multiplication)
- operator - (negation)
- operator - (subtraction)
- operator / (division)
- operator += (assignment)
- operator -= (assignment)
- operator *= (assignment)
- operator /= (assignment)
- operator == (equality)
- operator != (inequality)

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
```

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.

RELATED CONCEPTS

Complex Mathematics Library Overview (page 77)
Review of Complex Numbers (page 77)
Header Files and Constants for the complex and c_exception Classes (page 78)
Friend Functions for complex (page 80)
Input and Output Operators for complex (page 81)
Error Functions (page 81)

RELATED TASKS

Use Mathematical Operators for complex

Friend Functions for complex

The complex class defines a set of mathematical, trigonometric, magnitude, and conversion functions as friend functions of complex objects. They are:

- exp (exponent)
- log (natural logarithm)
- pow (power)
- sqrt (square root)
- cos (cosine)
- cosh (hyperbolic cosine)
- sin (sine)
- sinh (hyperbolic sine)
- abs (absolute value or magnitude)
- norm (square of magnitude)
- arg (polar angle)
- conj (conjugate)
- polar (polar to complex)
- real (real part)
- imag (imaginary part)

RELATED CONCEPTS

Complex Mathematics Library Overview (page 77)
Review of Complex Numbers (page 77)
Header Files and Constants for the complex and c_exception Classes (page 78)
Mathematical Operators for complex (page 79)
Input and Output Operators for complex (page 81)
Error Functions (page 81)

RELATED TASKS

Use Friend Functions with complex (page 84)

Input and Output Operators for complex

The complex class defines input and output operators for USL I/O Stream Library:

- operator >> (input)
- operator << (output)

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.

RELATED CONCEPTS

Complex Mathematics Library Overview (page 77)
Review of Complex Numbers (page 77)
Header Files and Constants for the complex and c_exception Classes (page 78)
Mathematical Operators for complex (page 79)
Friend Functions for complex (page 80)
Error Functions (page 81)

RELATED TASKS

Use complex Input and Output Operators (page 82)

Error Functions

There are three recommended methods to handle complex mathematics errors:

- use the c_exception class
- define a customized complex_error function
- handle errors outside of the complex mathematics library

Using the c_exception Class
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. If you do not define your own complex_error function, complex_error sets the complex return value and the errno error number.

Defining a Customized complex_error Function
You can either use the default version of complex_error() or define your own version of the function. If you define your own complex_error() function, and this function returns a nonzero value, no error message will be generated.

Handling Errors 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().

RELATED CONCEPTS

Complex Mathematics Library Overview (page 77)
Review of Complex Numbers (page 77)
Header Files and Constants for the complex and c_exception Classes (page 78)
Mathematical Operators for complex (page 79)
Friend Functions for complex (page 80)
Input and Output Operators for complex (page 81)

RELATED TASKS

Handle complex Mathematics Errors (page 86)

Construct complex Objects

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

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

RELATED CONCEPTS

Complex Mathematics Library Overview (page 77)
Header Files and Constants for the complex and c_exception Classes (page 78)

RELATED TASKS

Use complex Input and Output Operators (page 82)
Use Mathematical Operators for complex
Use Friend Functions with complex (page 84)

Use complex Input and Output Operators

The following example demonstrates the use of complex input and output operators:

```
// 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

int main(int argc, char *argv[]) {
  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:" << endl;
  cin >> b[0] >> b[1] >> b[2];
  cout << "Enter three more complex values:" << endl;
  cin >> c[2] >> c[0] >> c[1];

  // read input for scalar d
  cout << "Enter one more complex value:" << endl;
  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] << endl
       << b[0] << b[1] << b[2] << endl
       << c[1] << endl
       << "Here is scalar d: "
       << d << endl

       // cout << a produces an address, not a list of array elements:
       << "Here is the address of array a:" << endl
       << a
       << endl; //endl flushes the output stream
  return 0;
}
```

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:
0x2ff21cc0
```

RELATED CONCEPTS

Complex Mathematics Library Overview (page 77)

Header Files and Constants for the complex and c_exception Classes (page 78)
Input and Output Operators for complex (page 81)
USL I/O Streaming (page 61)

RELATED TASKS

Construct complex Objects (page 82)
Use Mathematical Operators for complex
Use Friend Functions with complex (page 84)
Combine Input and Output of Different Types
Receive Input from Standard Input
Display Output from Standard Output or Standard Error

Use Friend Functions with 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
```

Use Friend Functions for complex
The complex class defines four mathematical functions as friend functions of complex objects.

- 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>
int main(int argc, char *argv[])
{
   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;
   return 0;
   }
```

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)
```

Use Trigonometric Functions for complex

The complex class defines four trigonometric functions as friend functions of complex objects.

- 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>

int main(int argc, char *argv[])

{
   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;
return 0;
}
```

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)
```

Use Magnitude Functions for complex

The magnitude functions for complex are:

- abs - Absolute value
- norm - Square magnitude

Use 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 to real part
- imag - extract imaginary part

The following program shows how to use complex conversion functions:

```
// Using the complex conversion functions

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

int main(int argc, char *argv[])
{
   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;
   return 0;
}
```

This example produces the output shown below, 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)
```

RELATED CONCEPTS

Complex Mathematics Library Overview (page 77)
Header Files and Constants for the complex and c_exception Classes (page 78)
Friend Functions for complex (page 80)

RELATED TASKS

Construct complex Objects (page 82)
Use complex Input and Output Operators (page 82)
Use Mathematical Operators for complex

Handle complex Mathematics Errors

You can use one of the following methods to handle complex mathematics errors:

- use the c_exception class
- define a customized complex_error function
- compile a program that uses a customized complex_error function

Use c_exception to Handle complex Mathematics Errors
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 as follows:

```
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.

Define a Customized complex_error Function
You can either use the default version of complex_error() or define your own version of the function.

z/OS AIX When defining your own version of the complex_error() function, you must link your application to the static version of the complex library.

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;
 }

int main(int argc, char *argv[])
{
   complex c1(DBL_MAX,0);
   complex result;
   result = exp(c1);
   cout << "exp" << c1 << "= " << result << endl;
   return 0;
}
```

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

Compile 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 the complex_error is included in your source file when you compile you program.

RELATED CONCEPTS
Complex Mathematics Library Overview (page 77)
Header Files and Constants for the complex and c_exception Classes (page 78)
Error Functions (page 81)

Example: Calculate Roots

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 *theta*, and the distance between the origin and the point *z* is *r*. Then *z* has the polar form (*r*, *theta*), and the *n* roots of *z* have the values:

```
sigma
sigma x omega
sigma x omega^2
sigma x omega^3
.
.
.
sigma x omega^(n - 1)
```

where omega is a complex number with the value:

```
omega = (cos(2pi / n), sin(2pi / n))
```

and sigma 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 omega and sigma. The user is prompted for the complex value *z* and the value of *n*. After the values of omega and sigma 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;
   }
   int main(int argc, char *argv[])
   {
      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;
   }
   return 0
}
```

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)
```

RELATED CONCEPTS

Complex Mathematics Library Overview (page 77)
Review of Complex Numbers (page 77)
Header Files and Constants for the complex and c_exception Classes (page 78)

RELATED TASKS
Example: Use Equality and Inequality Operators (page 90)

Example: Use Equality and Inequality Operators

The functions is_equal and is_not_equal in the following example provide a reliable comparison between two complex values:

```
// 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);
}

int main(int argc, char *argv[])
{
    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;
      }
   return 0
   }
```

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     ~     ~    !=    ==
```

RELATED CONCEPTS

Complex Mathematics Library Overview (page 77)
Review of Complex Numbers (page 77)
Header Files and Constants for the complex and c_exception Classes (page 78)

RELATED TASKS

Example: Calculate Roots (page 88)

Chapter 9. Other Utilities

c++filt Name Demangling Utility

Note: This section applies to C++ programs only. C language programs do not use name mangling.

When VisualAge C++compiles a C++ program, it encodes all function names and certain other identifiers to include type and scoping information. This encoding process is called mangling. The linker uses these mangled names to ensure type-safe linkage. These mangled names appear in the object files and final executable file.

Tools that use these files, the AIX dump utility for example, have only the mangled names and not the original source-code names, and present the mangled name in their output. This output is undesirable because the names are difficult to read.

Two utilities convert the mangled names to their original source code names:

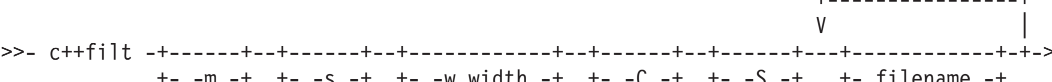
c++filt A filter that demangles (decodes) mangled names.
demangle.h A class library that you can use to develop tools to manipulate mangled names.

Demangling Compiled C++ Names with c++filt

The c++filt utility converts mangled names to demangled names. You can enter any mangled name and obtain the demangled name. The filter copies characters from the given file names or standard input to standard output, replacing all mangled names with their corresponding demangled names.

To use the c++filt utility, type /usr/vacpp/bin/c++filt followed by any options, on the shell command line.

Syntax of the c++filt Utility



```
                                                                          +----------------+
                                                                          V                |
>>- c++filt -+------+--+------+--+------------+--+------+--+------+---+------------+-+->
             +- -m -+  +- -s -+  +- -w width -+  +- -C -+  +- -S -+   +- filename -+
```

You can select one or more of the following options:

-m Produces a symbol map on standard output. This map contains a list of the mangled names and their corresponding demangled names. This output follows the filtered output.

-s	Produces a side-by-side demangling, with each mangled name encountered in the input stream replaced by a combination of the demangled name followed by the original mangled name.
-w *width*	Prints demangled names in fields width characters wide. If the name is shorter than width, it is padded on the right with blanks; if longer, it is truncated to width.
-C	Demangles stand-alone class names such as Q2_1X1Y.
-S	Demangles special compiler generated symbol names such as __vft1X.
filename	Is the name of the file containing the mangled names you want to demangle. You can specify more than one filename.

The following example shows the symbols contained in an object file functions.o, producing a side-by-side listing of the mangled and demangled names with a field width of 40 characters:

```
dump -tv functions.o | c++filt -s -w 40
```

Demangling Compiled C++ Names with the demangle Class Library

The demangle class library contains a small class hierarchy that client programs can use to demangle names and examine the resulting parts of the name. It also provides a C-language interface for use in C programs. Although it is a C++ library, it uses no external C++ features so you can link it directly to C programs. Demangle is included as part of libC.a, and is automatically linked, when required, if libC.a is linked.

You can write programs that use demangle.h to take a mangled name and return the demangled name, and the separate parts of the name. For example, given the mangled name of a member function, you can:

- Get the text of the entire demangled name
- Get the text of the function name
- Get the text of the qualifier list and each of its parts
- Get the text of the parameter list and each of its parts
- Ask questions about whether the function is const or volatile

Using demangle.h in C++ Programs

To demangle a name (represented as a character array), create a dynamic instance of Name class, providing the character string to the class's constructor. For example, if the compiler mangled X::f(int) to the mangled name f__1XFi, in order to demangle the name, enter:

```
char *rest;
Name *name = Demangle("f__1XFi", rest) ;
```

The demangler classifies names into four categories: functions, member functions, variables, and member variables. Once your program constructs an instance of class Name, your program can ask what kind of Name the instance is, using the Kind method of Name. Based on the kind of name returned, the program can ask for the text of the different parts of the name, or the text of the entire name.

For the mangled name f__1XFi example, you can find out the following:

```
name->Kind() == MemberFunction
((MemberFunctionName *)name)->Scope()->Text() is "X"
((MemberFunctionName *)name)->RootName() is "f"
((MemberFunctionName *)name)->Text() is "X::f(int)"
```

If the supplied character string is not a name that requires demangling, because the original name was not mangled, the Demangle function returns NULL.

For further details about the demangle.h library and the C++ interface, look at the comments in the library's header file, /usr/vacpp/include/demangle.h.

CreateExportList Command

Syntax

CreateExportList [-r] *exp-listname* [-f *filelistname* | *obj_files*] [-X 32|64]

where:
exp-listname is the file name that contains the list of global symbols found in the object files specified by either *obj_files* or -f *filelistname*.

obj_files is actual names of object files.

filelistname is the file that contains a list of object filenames.

Description
This command creates a file containing a list of all the global symbols found in a given set of object files. Template prefixes are prunned, and `__rsrc`ignored if **-r** is specified. The **-f** option specifies a file containing the list of object files. The first argument is the export list file name (which is overwritten) and the remaining arguments are the object files.

CreateExportList creates an empty list if any of the following are true:
- no object files are given
- the -f option is missing
- the file specified by -f is empty.

Suboptions

-r	Do not add resource file symbol (__rsrc) to the exports list (but still export it).
-X 32|64	Generate names from 32-bit or 64-bit object files in input list. -X32 is the default.

RELATED CONCEPTS
Constructing a Library (page 96)

RELATED TASKS
Initialize Shared Library (page 102)

RELATED REFERENCES
mkshrobj Batch Compiler

linkxlC Command

linkxlC is a small shell script that links C++ .o and .a files. It can be redistributed and used by someone who does not have VisualAge C++ installed. It runs the **munch** utility, and then invokes **ld**.

The script supports linker options. It ignores most other options.

For a full description of the **ld** command, refer to the AIX Version 4 Commands Reference.

RELATED REFERENCES
Compiler Options

Constructing a Library

Libraries with Non-Shared Files
Non-shared files are files that are linked into the C++ executable program.

Construct a library using these steps:

- Compile each file using the **-qpriority=***N* compiler option and if applicable in your application, **#pragma priority**(*N*) directives within the file. Normally, simply specifying the priority level for the file with the -qpriority=N option is sufficient. Use numbers for *N* to specify the priority levels at which you want objects initialized.
- Use the AIX **ar** command to link the files and produce an archive library file.

```
xlC -c -o bar.o example.C ar rv libfoo.a bar.o
```

Libraries with Shared Files
Shared files are files that are used by more than one program.

You should compile with the **-qmkshrobj** compiler option or use the **makeC++SharedLib** program found in /usr/vacpp/bin and the AIX **ar** command.

```
xlC -c example.C xlC -qmkshrobj -o foo.o ar rv libfoo.a foo.o bar.o
```

RELATED TASKS

Intialize Shared Library (page 102)
Specify Priority Levels for Library Objects (page 103)
Example of Object Initialization in a Group of Files (page 104)

RELATED REFERENCES

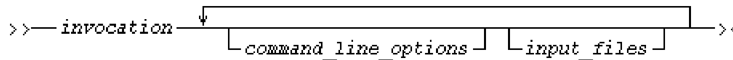
makeC++SharedLib Command (page 97)
-qmkshrobj Compiler Option

makeC++SharedLib Command

makeC++SharedLib is a shell script that links C++ .o and .a files. It can be redistributed and used by someone who does not have VisualAge C++ installed.

We recommend that you use the -qmkshrobj batch compiler option instead of the **makeC++SharedLib** command. The advantage to using this option is that the compiler will automatically include and compile the template instantiations in the tempinc directory.

This page describes the **makeC++SharedLib** command and provides an example of how to use it to make two shared libraries. It describes how to combine these two files in a library using the **ar** command so that object initialization takes place in the specified order.



Command Line Options

invocation	Is the path name for the **makeC++SharedLib** command that constructs the shared library file.
-o*shared_file*.o	Is the name of the file that will hold the shared file information. The default is shr.o.
-bOptions	Uses the **-b** binder options of the **ld** command.
-L*lib_dir*	Uses the **-L** option of the **ld** command to add the directory *lib_dir* to the list of directories to be searched for unresolved symbols. The **ld** command is described in the AIX Version 4 Commands Reference.
-l*library*	Adds *library* to the list of libraries to be searched for unresolved symbols.
-p *priority*	Specifies the priority level for the file. *priority* may be any number from -214782623 (highest priority-initialized first) to 214783647 (lowest priority-initialized last). Numbers from -214783648 to -214782624 are reserved for system use.
-I *import_list*	Uses the **-bI** option of the **ld** command to resolve the list of symbols in the file *import_list* that can be resolved by the binder.
-E *export_list*	Uses the **-bE** option of the **ld** command to export the external symbols in the export_list file. If you do not specify **-E** *export_list*, a list of all global symbols is generated.

-e *file*	Saves in file the list computed by **-E** *export_list*.
-n *name*	Sets the entry name for the shared executable to name. This is equivalent to using the command **ld -e** *name*
-X *mode*	Specifies the type of object file makeC++SharedLib should create. The mode must be either 32, which processes only 32-bit object files, or 64, which processes only 64-bit object files. The default is to process 32-bit object files (ignore 64-bit objects). The mode can also be set with the OBJECT_MODE environment variable. For example, OBJECT_MODE=64 causes makeC++SharedLib to process any 64-bit objects and ignore 32-bit objects. The -X flag overrides the OBJECT_MODE variable.

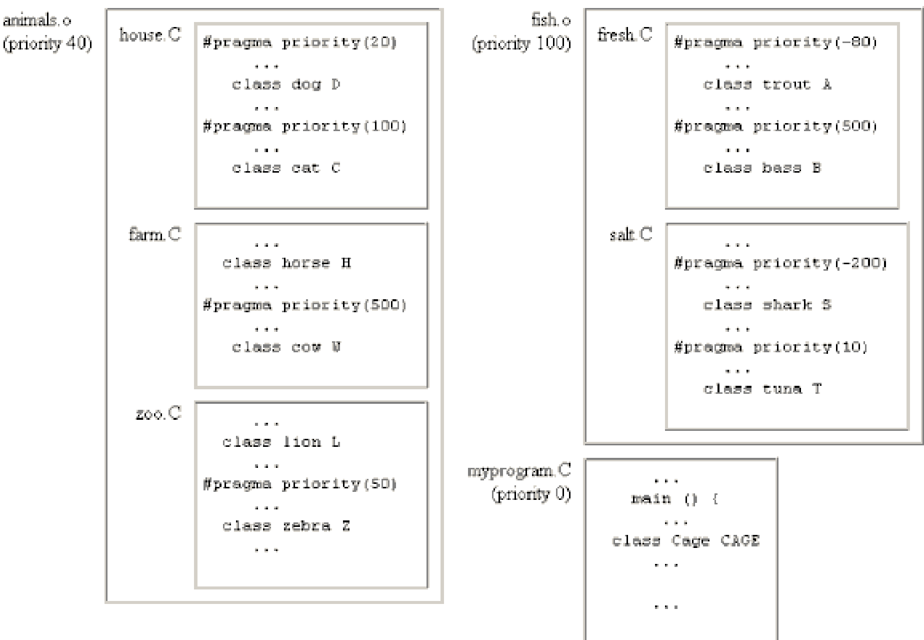
Input Files

file.o	Is an object file to be put into the shared library.
file.a	Is an archive file to be put into the shared library.

Example

The following example shows how to construct two shared libraries using the **makeC++SharedLib** command, and then use the AIX **ar** command to combine these libraries along with a file that contains the main function so that objects are initialized in the specified order.

The drawing below shows how the objects in this example are arranged in various files.



The first part of this example shows how to use **makeC++SharedLib** along with the **-qpriority=***N* option and the **#pragma priority**(*N*) directive to specify the initialization order for objects in these files.

The example shows how to make two shared libraries: animals.o containing object files compiled from house.C, farm.C, and zoo.C, and fish.o containing object files compiled from fresh.C and salt.C.

The example shows how to specify priorities and use the ar command so that all the objects in fish.o are initialized before the objects in myprogram.o, and all the objects in animals.o are initialized after the objects in myprogram.o. Within animals.o, the objects in zoo.C are initialized before the objects in house.C and farm.C.

To specify this initialization order, follow these steps:

1. Develop an initialization order for the objects in house.C, farm.C, and zoo.C:
   a. To ensure that the object lion L in zoo.C is initialized before any other objects in either of the other two files, compile zoo.C using a **-qpriority=***N* option with *N* less than zero so both objects have a priority number less than any other objects in farm.C and house.C:
      ```
      xlC zoo.C -c -qpriority=-50
      ```
   b. Compile the house.C and farm.C files without specifying the **-qpriority=***N* option (so *N*=0) so objects within the files retain the priority numbers specified by their **#pragma priority**(*N*) directives:
      ```
      xlC house.C farm.C -c
      ```
   c. Combine these three files in a shared library. Use **makeC++SharedLib** to construct a library animals.o with a priority of 40:
      ```
      makeC++SharedLib -o animals.o -p 40 house.o farm.o zoo.o
      ```
2. Develop an initialization order for the objects in fresh.C, and salt.C:
   a. Compile the fresh.C and salt.C files:
      ```
      xlC fresh.C salt.C -c
      ```

b. To assure that all objects in fresh.C and salt.C are initialized before any other objects, use **makeC++SharedLib** to construct a library fish.o with a priority of -100.

```
makeC++SharedLib -o fish.o -p -100 fresh.o salt.o
```

Because the shared library fish.o has a lower priority number (-100) than animals.o (40), when the files are placed in an archive file with the **ar** command, their objects are initialized first.

3. Compile myprogram.C that contains the function main to produce an object file myprogram.o. By not specifying a priority, this file is compiled with a default priority of zero, and the objects in main have a priority of zero.

```
xlC myprogram.C -c
```

4. To create a library that contains the two shared libraries, and the program myprogram.o that contains the function main, so that the objects are initialized in the order you have specified, you use the **ar** command. To produce an archive file, prio_lib.a, enter the command:

```
ar rv prio_lib.a animals.o fish.o myprogram.o
```

where:

rv
: Are two **ar** options. r replaces a named file if it already appears in the library, and v writes to standard output a file-by-file description of the making of the new library.

prio_lib.a
: Is the name you specified for the archive file that will contain the shared library files and their priority levels.

animals.o
fish.o
: Are the two shared files you created with **makeC++SharedLib**.

myprogram.o
: Is the name of the file that contains the function main.

The order of initialization of the objects is shown in the following table.

Order of Initialization of Objects in priolib.a			
File	**Class Object**	**Priority Value**	**Comment**
“fish.o”		-100	All objects in “fish.o” are initialized first because they are in a library prepared with **makeC++SharedLib** -p -100 (lowest priority number, -p -100, specified for any files in this compilation)
	“shark S”	-100(-200)	Initialized first in “fish.o” because within file, #pragma priority(-200)
	“trout A”	-100(-80)	#pragma priority(-80)
	“tuna T”	-100(10)	#pragma priority(10)
	“bass B”	-100(500)	#pragma priority(500)

Order of Initialization of Objects in priolib.a			
"myprog.o"		0	File generated with no priority specifications; default is 0
	"CAGE"	0(0)	Object generated in **main** with no priority specifications; default is 0
"animals.o"		40	File generated with **makeC++SharedLib** with -p 40
	"lion L"	40(-50)	Initialized first in file "animals.o" compiled with -qpriority=-50
	"horse H"	40(0)	Follows with priority of 0 (since -qpriority=*N* not specified at compilation and no #pragma priority(*N*) directive)
	"dog D"	40(20)	Next priority number (specified by #pragma priority(20))
	"zebra N"	40(50)	Next priority number from #pragma priority(50)
	"cat C"	40(100)	Next priority number from #pragma priority(100)
	"cow W"	40(500)	Next priority number from #pragma priority(500) (Initialized last)

5. To produce an executable file, animal_time, so that the objects are initialized in the order you have specified, enter:

```
xlC prio_lib.a -oanimal_time
```

You can place both nonshared and shared files with different priority levels in the same archive library using the AIX **ar** command.

RELATED CONCEPTS

Constructing a Library (page 96)

RELATED TASKS

Initialize Shared Library (page 102)

RELATED REFERENCES

mkshrobj Compiler Option

Initialize Shared Library

In some C++ programs, it is important to specify the order in which objects are initialized.

Before the main function of a C++ program is executed, the language definition ensures that all objects with constructors from all the files included in the C++ program have been properly constructed. The language definition, however, does not specify the order of initialization for these objects across files. In some cases, you may want to specify the initialization order of some objects in your program.

Often, your program will be made up of several files and files contained in libraries. The libraries that you use with your C++ source program may contain object (.o) files that have components shared with other programs (shared files), as well as files that are only used by your program (non-shared files).

To specify an initialization order, you can:
- Specify an initialization priority number for objects within a file using the #pragma priority directive.
- Generate shared objects using the -qmkshrobj compiler option, then construct an archive (.a) library containing several shared and non-shared objects.

The run-time environment initializes the objects in these libraries in the order of their priority number. Priority numbers can range from -2147483648 to 2147483647. However, numbers from -2147483648 to -2147482624 are reserved for system use. The highest priority you can specify is -2147482623, and it is initialized first. The lowest priority, 2147483647, is initialized last.

Objects in files with identical priorities are initialized in random order.

When your program exits, the destructors for global and static objects are invoked in the reverse order of their construction.

If you do not specify priority levels, the default priority is 0 (zero). If there are multiple shared objects with different priority levels, the priority levels determine the order in which they will be initialized. Within a single shared object or the main function, #pragma priority controls the initialization order. The executable program has a priority of 0.

The loadAndInit (page 105) routine will initialize shared libraries. Likewise, terminateAndUnload (page 106) will terminate them. Include the file load.h to use these routines, which are found in the libC.a library.

RELATED CONCEPTS

Constructing a Library (page 96)

RELATED TASKS

Specify Priority Levels for Library Objects (page 103)
Example of Object Initialization in a Group of Files (page 104)

RELATED REFERENCES

makeC++SharedLib Command (page 97)
mkshrobj Compiler Option

Specify Priority Levels for Library Objects

These examples are intended to show how you can specify priority levels for objects within a file, at the file level, and at the library level. However, in most applications it is not necessary to specify more than one or two priority levels.

Specifying Priority Levels within a File
To specify the order of initialization of objects within a file, use the #pragma priority directive. You can use any number of directives within the file, but the priority numbers must be in increasing order. That is, you cannot specify an object with a smaller priority number after you have specified one with a larger priority number.

The following example shows how to specify the priority for several objects within a source file.

```
...
#pragma priority(5) //Following objects constructed with priority 5
...
static struct base A ;
class house B ;
...
#pragma priority(10) //Following objects constructed with priority 10
...
class barn C ;
...
#pragma priority(2) // Error - priority number must be larger
// than preceding number (10)
...
#pragma priority(20) //Following objects constructed with priority 20
...
class garage D ;
...
```

Specifying the Priority Level of a File
To specify the priority level of a file, use the -qpriority compiler option. Use this option if you want all the objects in the file to have the same priority level, and you do not want to write #pragma priority(*N*) directives in the file.

For example, using the batch compiler option -qpriority=20, is equivalent to using #pragma priority(20).

If there are no #pragma priority directives within the file, all objects within the file have the priority specified with -qpriority= .

If there are #pragma priority directives within the file, all objects found within the file up to the first #pragma priority directive are given the same priority number as specified for the file. The objects after a #pragma priority directive are given that priority number of *N* until the next #pragma priority directive is encountered.

Within the file, the first #pragma priority must have a higher priority number than the number used in the -qpriority option and subsequent #pragma priority directives must have increasing numbers.

RELATED TASKS
Initialize Shared Library (page 102)
Example of Object Initialization in a Group of Files (page 104)

Example of Object Initialization in a Group of Files

You can specify different priority numbers for objects within files, and the compiler will initialize them in the following order:

1. #pragma priority
2. By line and column within a file

The following example describes describes the initialization order for objects in two files, farm.C and zoo.C. Both files use #pragma priority directives. The following table shows part of the files with #pragma priority directives and hypothetical objects:

farm.C

```
#pragma priority(20)
...
class dog A ;
class dog B ;
...
#pragma priority(100)
...
class cat C ;
class cow D ;
...
#pragma priority(200)
class mouse E ;
...
```

zoo.C

```
...
class lion K ;
#pragma priority(30)
class bear M ;
...
#pragma priority(50)
...
class zebra N ;
class snake S ;
...
#pragma priority(250)
class frog F ;
...
```

Compile farm.C and zoo.C with -qpriority=10.

Initialization takes place in the following order:

Object	Priority Value	Comment
"lion K"	10	Takes priority number of file "zoo.o" (10) (Initialized first)
"dog A"	20	Takes #PRAGMA PRIORITY(20) priority.
"dog B"	20	Follows "dog A"
"bear M"	30	Next priority number, specified by #PRAGMA PRIORITY(30)
"zebra N"	50	Next priority number from #PRAGMA PRIORITY(50)
"snake S"	50	Follows with same priority
"cat C"	100	Next priority number
"cow D"	100	Follows with same priority
"mouse E"	200	Next priority number
"frog F"	250	Next priority number (Initialized last).

RELATED TASKS

Initialize Shared Library (page 102)

RELATED REFERENCES

-qpriority Compiler Option
-qmkshrobj Compiler Option

loadAndInit Routine

Format

```
int (*loadAndInit(char *FilePath, unsigned int Flags, char *LibraryPath))();
```

Description

The **loadAndInit()** routine calls the AIX **load** subroutine to load the specified module (shared library) into the calling process's address space. If the shared library is loaded successfully, any C++ initialization is performed. The **loadAndInit()** routine ensures that a shared library is only initialized once. Subsequent loads of the same shared library will not perform any initialization of the shared library.

If loading a shared library results in other shared libraries being loaded, the initialization for those shared libraries will also be performed (if it has not been previously). If loading a shared library results in the initialization of multiple shared libraries, the order of initialization is determined by the priority assigned to the shared libraries when they were built. Shared libraries with the same priority are initialized in random order.

Use the terminateAndUnload() (page 106) subroutine to unload the shared library instead of the unload routine.

Do not reference symbols in the C++ initialization that need to be resolved by a call to the **loadbind** routine since the **loadbind** routine normally is not called until after the **loadAndInit** routine returns. In the rare cases where the **loadbind**routine must be called before the C++ initialization is performed one can use the following general steps to load and initialize the shared library. Load the shared library with the **load** routine call the **loadbind** routine to resolve the symbols. load and initialize the shared library with the **loadAndInit** routine. unload the shared library with the **unload** routine. This step is required to perating system unloads the shared library when the **terminateAndUnload** routine is called.

In previous versions of the compiler, the **loadAndInit** routine had to be called at least once before on a different shared library before these four steps would work. If the **loadquery** routine can be used to determine which module to bind the symbols too, an alternative approach is to add a C++ initialization that will call the **loadquery**and **loadbind** routines. The priority of this C++ initialization has to be high enough so that it is performed before the real C++ initialization is performed.

Parameters

FilePath Points to the name of the shared library being loaded. The name may be a base name (doesn't contain any '/' characters) or a relative or full path name (contains a '/'). No search is performed for relative or full path names.

Flags Modifies the behaviour of load. If no special behaviour is required, set the value to 0 (or 1). The possible flags are:

- **L_LIBPATH_EXEC** - specifies that the library path used at process exec time be prepended to any library path specified in the loadAndInit call. You shoul use this flag.
- **L_NOAUTODEFER** - specifies that any deferred imports must be explicitly resolved by the use of the **loadbind** subroutine.
- **L_LOADMEMBER** - specifies that the *FilePath* is the name of a member in an archive. The format is `libfoo.a(member)`.
- *LibraryPath* Points to the default library search path.

Return Values

Upon successful completion, the loadAndInit subroutine returns the pointer to function for the entry point (or data section) of the shared library.

If the **loadAndInit** subroutine fails, a null pointer is returned, the module is not loaded or initialized, and the **errno** global variable is set to indicate the error. The possible errors are the same as the **load** subroutine.

RELATED REFERENCES

terminateAndUnload (page 106) subroutine
-qmkshrobj Compiler Option
The AIX operating system subroutines **load**, **dlopen**, **loadbind**, **loadquery**, and **unload**.

terminateAndUnload Routine

Format

```
#include <load.h>
int terminateAndUnload(int (*FunctionPointer)());
```

Description

The **terminateAndUnload** subroutine performs any C++ termination that is required and unloads the module (shared library). The value returned by the **loadAndInit** subroutine is passed to the **terminateAndUnload** subroutine as *FunctionPointer*. If this is the last time the shared library is being unloaded, any C++ termination is performed for this shared libary and any other shared libraries that are being unloaded for the last time as well. The order of termination is the reverse order of initialization performed by the **loadAndInit**subroutine. If any uncaught exceptions occur during the C++ termination the termination is stopped and the shared library is unloaded.

If the **loadAndInit** subroutine is called more times for a shared library than **terminateAndUnload**, the shared library will never have the C++ termination peformed. If you rely on the C++ termination being performed at the time the **terminateAndUnload** subroutine is called, ensure the number of calls to the **terminateAndUnload** subroutine match the number of calls to the **loadAndInit** subroutine. If any shared libraries loaded with the **loadAndInit** subroutine are still in use when the program exits, the C++ termination is performed.

If the **terminateAndUnload** subroutine is used to unload shared libraries not loaded with the **loadAndInit** subroutine, no termination will be performed.

The **terminateAndUnload** subroutine takes the same parameters and returns the same values and error codes as the **unload** subroutine. See the **unload** subroutine in your AIX documentation for more information.

Parameters

FunctionPointer Specifies the name of the function returned by the **loadAndInit** subroutine.

Return Values

Successful completion of the **terminateAndUnload** subroutine returns a value of `0`, even if the C++ termination was not performed and the shared library was not unloaded because the shared library was still in use.

Error Codes

If the **terminateAndUnload** subroutine fails, it returns a value of `-1`and sets **errno** to indicate the error.

RELATED REFERENCES

loadAndInit (page 105) Routine
-qmkshrobj Compiler Option
The AIX operating system subroutines **load**, **dlopen**, **loadbind**, **loadquery**, and **unload**.

_makepath — Create Path

Format

```
#include <extension.h>
void _makepath(char *path, char *drive, char *dir,
                 char *fname, char *ext);
```

Language Level: Extension
_makepath creates a single path name, composed of a directory path, file name, and file name extension.

The *path* argument should point to an empty buffer large enough to hold the complete path name. The constant FILENAME_MAX, defined in <stdio.h>, specifies the maximum size allowed for *path*. The other arguments point to the following buffers containing the path name elements:

dir	The path of directories, not including the drive designator or the actual file name. The trailing slash is optional, and can be used in a single dir argument. If a trailing slash is not specified, it is inserted automatically. If dir is a null character or an empty string, no slash is inserted in the composite *path* string.
fname	The base file name without a suffix.
ext	The actual file name suffix, with or without a leading period. _makepath inserts the period automatically if it does not appear in ext. If ext is a null character or an empty string, no period is inserted in the composite path string.

The size limits on the above four fields are those specified by the constants _MAX_DRIVE, _MAX_DIR, _MAX_FNAME, and _MAX_EXT, which are defined in <extension.h>. The composite path should be no larger than the FILENAME_MAX constant defined in <stdio.h>; otherwise, the operating system does not handle it correctly.

Note: No checking is done to see if the syntax of the file name is correct.

Return Value
There is no return value.

Example
This example builds a file name path from the specified components.

```
#include <stdio.h>
#include <extension.h>

int main(void)
{
   char path_buffer[_MAX_PATH];
   char dir[_MAX_DIR];
   char fname[_MAX_FNAME];
   char ext[_MAX_EXT];

   _makepath(path_buffer, "qc//bob//eclibref//e", "makepath", "c");
   printf("Path created with _makepath: %s/n/n", path_buffer);
   _splitpath(path_buffer, drive, dir, fname, ext);
   printf("Path extracted with _splitpath:/n");
   printf("directory: %s/n", dir);
   printf("file name: %s/n", fname);
   printf("extension: %s/n", ext);
   return 0;

   /*******************************************************************
      The output should be:

      Path created with _makepath: qc/bob/eclibref/e/makepath.c

      Path extracted with _splitpath:
      directory: qc/bob/eclibref/e/
      filename: makepath
      extension: .c
   *******************************************************************/
}
```

RELATED REFERENCES
_fullpath — Get Full Path Name of Partial Path
_splitpath — Decompose Path Name (page 108)

_splitpath — Decompose Path Name

Format

```
#include <extension.h>
void _splitpath(char *path, char *drive, char *dir,
                    char *fname, char *ext);
```

Language Level: Extension
_splitpath decomposes an existing path name *path* into its four components. The *path* should point to a buffer containing the complete path name.

The maximum size necessary for each buffer is specified by the _MAX_DRIVE, _MAX_DIR, _MAX_FNAME, and _MAX_EXT constants defined in <extension.h>. The other arguments point to the following buffers used to store the path name elements:

Buffer	Description
dir	Contains the path of subdirectories, if any, including the trailing slash. Slashes (/) may be present in *path*.
fname	Contains the base file name without any extensions.
ext	Contains the file name extension, if any, including the leading period (.).

You can specify NULL for any of the buffer pointers to indicate that you do not want the string for that component returned.

The return parameters contain empty strings for any path name components not found in *path*.

Return Value
There is no return value.

Example
This example builds a file name path from the specified components, and then extracts the individual components.

```
#include <stdio.h>
#include <extension.h>

int main(void)
{
   char path_buffer[_MAX_PATH];
   char dir[_MAX_DIR];
   char fname[_MAX_FNAME];
   char ext[_MAX_EXT];

   _makepath(path_buffer, "qc\\bob\\eclibref\\e", "makepath", "c");
   printf("Path created with _makepath: %s\n\n", path_buffer);
   _splitpath(path_buffer,dir, fname, ext);
   printf("Path extracted with _splitpath:\n");
   printf("directory: %s\n", dir);
   printf("file name: %s\n", fname);
   printf("extension: %s\n", ext);
   return 0;

   /*******************************************************************
      The output should be:

      Path created with _makepath: qc\bob\eclibref\e\makepath.c

      Path extracted with _splitpath:
      directory: qc\bob\eclibref\e\
      file name: makepath
      extension: .c
   *******************************************************************/
}
```

RELATED REFERENCES

_fullpath — Get Full Path Name of Partial Path
_makepath — Create Path (page 107)

Appendix. Non-ISO USL Classes

complex

This class provides you with facilities to manipulate 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.

Class header file: complex.h

complex - Hierarchy List

complex

complex - Member Functions and Data by Group

Constructors & Destructor

These constructors can be used to create complex objects.

There is no explicit complex destructor.

Arrays of Complex Numbers

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

In the following example, the elements of array *b* are initialized to the values in the initial value list, but only the real parts of elements of array *a* are initialized to the values in the initial value list.

```
#include < complex.h >

int main()
{
complex a[3] = {1.0, 2.0, 3.0};
complex b[3] = {complex(1.0, 1.0), complex(2.0, 2.0), complex(3.0, 3.0)};

cout <<  "Here is the first element of a:  " << a[0] << endl;
cout <<  "Here is the first element of b:  " << b[0] << endl;
}
```

This example produces the following output:

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

complex

Constructs a complex number.

Overload 1

```
public:complex(double r, double i = 0.0)
```

© Copyright IBM Corp. 1998, 2002

Constructs a complex number.

The first argument, r, is assigned to the real part of the complex number. If you specify a second argument, it is assigned to the imaginary part of the complex number. If the second parameter is not specified, the imaginary part is initialized to 0.

Overload 2

```
public:complex()
```

Constructs a complex number . The real and imaginary parts of the complex number are initialized to (0, 0).

Assignment Operators

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

```
complex x, y, z; // valid declaration

x = (y += z); // invalid assignment causes a compile-time error

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

operator *=

Assigns the value of x * y to x.

Overload 1

```
public:void operator *=(const complex&)
```

Overload 2

```
public:inline void operator *=(complex)
```

operator +=

Assigns the value of x + y to x.

Overload 1

```
public:inline void operator +=(complex)
```

Overload 2

```
public:inline void operator +=(const complex&)
```

operator -=

Assigns the value of x - y to x.

Overload 1

```
public:inline void operator -=(complex)
```

Overload 2

```
public:inline void operator -=(const complex&)
```

operator /=

Assigns the value of x / y to x.

Overload 1

```
public:inline void operator /=(complex)
```

Overload 2

```
public:void operator /=(const complex&)
```

complex - Associated Globals

abs

```
double abs(complex)
```

Returns the absolute value or magnitude of its argument. The absolute value of a complex value (a, b) is the positive square root of $a^2 + b^2$.

abs

```
double abs(const complex&)
```

Returns the absolute value or magnitude of its argument. The absolute value of a complex value (a, b) is the positive square root of $a^2 + b^2$.

arg

```
double arg(complex)
```

Returns the angle (in radians) of the polar representation of its argument. If the argument is equal to the complex number (a, b), the angle returned is the angle in radians on the complex plane between the real axis and the vector (a, b). The return value has a range of -pi to pi.

arg

```
double arg(const complex&)
```

Returns the angle (in radians) of the polar representation of its argument. If the argument is equal to the complex number (a, b), the angle returned is the angle in radians on the complex plane between the real axis and the vector (a, b). The return value has a range of -pi to pi.

conj

```
complex conj(complex)
```

Returns the complex value equal to (a, -b) if the input argument is equal to (a, b).

conj

```
inline complex conj(const complex&)
```

Returns the complex value equal to (a, -b) if the input argument is equal to (a, b).

cos

```
complex cos(complex)
```

Returns the cosine of the complex argument.

cos

```
complex cos(const complex&)
```

Returns the cosine of the complex argument.

cosh

```
complex cosh(complex)
```

Returns the hyperbolic cosine of the complex argument.

cosh

```
complex cosh(const complex&)
```

Returns the hyperbolic cosine of the complex argument.

exp

```
complex exp(complex)
```

Returns the complex value equal to *e* to the power of *x* where *x* is the argument.

exp

```
complex exp(const complex&)
```

Returns the complex value equal to *e* to the power of *x* where *x* is the argument.

imag

```
double imag(const complex&)
```

Extracts the imaginary part of the complex number provided as the argument.

imag

```
inline double imag(const complex&)
```

Extracts the imaginary part of the complex number provided as the argument.

log

```
complex log(complex)
```

Returns the natural logarithm of the argument *x*.

log

```
complex log(complex)
```

Returns the natural logarithm of the argument *x*.

norm

```
double norm(complex)
```

Returns the square of the magnitude of its argument. If the argument *x* is equal to the complex number (a, b), norm() returns the value $a^2 + b^2$.

norm() is faster than abs(), but it is more likely to cause overflow errors.

norm

```
double norm(const complex&)
```

Returns the square of the magnitude of its argument. If the argument *x* is equal to the complex number (a, b), norm() returns the value $a^2 + b^2$.

norm() is faster than abs(), but it is more likely to cause overflow errors.

operator !=

```
int operator !=(complex, complex)
```

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

Because both components are double values, the inequality operator returns false only when both the real and imaginary components of the two values are identical. If you want an inequality operator that can test for an absolute difference within a certain tolerance between the two pairs of corresponding components, you can use a function such as the *is_not_equal* function.

operator !=

```
inline int operator !=(const complex&, const complex&)
```

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

Because both components are double values, the inequality operator returns false only when both the real and imaginary components of the two values are identical. If you want an inequality operator that can test for an absolute difference within a certain tolerance between the two pairs of corresponding components, you can use a function such as the *is_not_equal* function.

operator *

```
complex operator *(complex, complex)
```

The multiplication operator returns the product of x and y.

This operator has the same precedence as the corresponding real operator.

operator *

```
complex operator *(const complex&, double)
```

The multiplication operator returns the product of x and y.

This operator has the same precedence as the corresponding real operator.

operator *

```
complex operator *(const complex&, const complex&)
```

The multiplication operator returns the product of x and y.

This operator has the same precedence as the corresponding real operator.

operator +

```
complex operator +(complex, complex)
```

The addition operator returns the sum of x and y.

This operator has the same precedence as the corresponding real operator.

operator +

```
inline complex operator +(const complex&, const complex&)
```

The addition operator returns the sum of x and y.

This operator has the same precedence as the corresponding real operator.

operator -

```
inline complex operator -(const complex&, const complex&)
```

The subtraction operator returns the difference between x and y.

This operator has the same precedence as the corresponding real operator.

operator -

```
complex operator -(complex, complex)
```

The subtraction operator returns the difference between x and y.

This operator has the same precedence as the corresponding real operator.

operator -

```
inline complex operator -(const complex&)
```

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

This operator has the same precedence as the corresponding real operator.

operator -

```
complex operator -(complex)
```

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

This operator has the same precedence as the corresponding real operator.

operator /

```
complex operator /(const complex&, double)
```

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

This operator has the same precedence as the corresponding real operator.

operator /

```
complex operator /(const complex&, const complex&)
```

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

This operator has the same precedence as the corresponding real operator.

operator /

```
complex operator /(complex, complex)
```

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

This operator has the same precedence as the corresponding real operator.

operator ==

```
int operator ==(complex, complex)
```

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

Because both components are double values, the equality operator tests for an *exact* match between the two sets of values. If you want an equality operator that can test for an absolute difference within a certain tolerance between the two pairs of corresponding components, you can use a function such as the *isequal* function.

operator ==

```
inline int operator ==(const complex&, const complex&)
```

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

Because both components are double values, the equality operator tests for an *exact* match between the two sets of values. If you want an equality operator that can test for an absolute difference within a certain tolerance between the two pairs of corresponding components, you can use a function such as the *isequal* function.

polar

```
complex polar(double, double = 0)
```

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

pow

```
complex pow(complex, double)
```

Returns the complex value x^y, where x is the first argument and y is the second argument.

pow

```
complex pow(double, complex)
```

Returns the complex value x^y, where x is the first argument and y is the second argument.

pow

```
complex pow(complex, complex)
```

Returns the complex value x^y, where x is the first argument and y is the second argument.

pow

```
complex pow(complex, int)
```

Returns the complex value x^y, where x is the first argument and y is the second argument.

pow

```
complex pow(const complex&, int)
```

Returns the complex value x^y, where x is the first argument and y is the second argument.

pow

```
complex pow(const complex&, double)
```

Returns the complex value x^y, where x is the first argument and y is the second argument.

pow

```
complex pow(const complex&, const complex&)
```

Returns the complex value x^y, where x is the first argument and y is the second argument.

pow

```
complex pow(double, const complex&)
```

Returns the complex value x^y, where x is the first argument and y is the second argument.

real

```
double real(const complex&)
```

Extracts the real part of the complex number provided as the argument.

real

```
inline double real(const complex&)
```

Extracts the real part of the complex number provided as the argument.

sin

```
complex sin(const complex&)
```

Returns the sine of the complex argument.

sin

```
complex sin(complex)
```

Returns the sine of the complex argument.

sinh

```
complex sinh(const complex&)
```

Returns the hyperbolic sine of the complex argument.

sinh

```
complex sinh(complex)
```

Returns the hyperbolic sine of the complex argument.

sqrt

```
complex sqrt(complex)
```

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

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

has two square roots:

```
(c, d)
(-c, -d)
```

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

complex - Inherited Member Functions and Data

Inherited Public Functions

None

Inherited Public Data

None

Inherited Protected Functions

None

Inherited Protected Data

None

---

c_exception

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

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

Class header file: complex.h

c_exception - Hierarchy List

c_exception

c_exception - Member Functions and Data by Group

Constructors & Destructor

You can construct objects of this class.

There is no explicit c_exception destructor.

c_exception

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

Construct a c_exception object.

n The name of the function where the error occurred.

a1 The first complex argument with which the function that caused the error was called.

a2 The second complex argument with which the function that caused the error was called.

c_exception - Associated Globals

complex_error

```
int complex_error(c_exception&)
```

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

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

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

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

These constants are defined in errno.h

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

Default Error-Handling Procedures

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

The following symbols appear in this table:

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

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

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

Note: errno is set to EDOM when the error for log() is detected.

Errors Not Handled

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

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

DBL_MAX is the maximum valid double value. INT_MAX is the maximum int value. Both these constants are defined in float.h.

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

cosh

```
complex cosh(complex)
```

Returns the hyperbolic cosine of the complex argument.

cosh

```
complex cosh(const complex&)
```

Returns the hyperbolic cosine of the complex argument.

exp

```
complex exp(complex)
```

Returns the complex value equal to *e* to the power of *x* where *x* is the argument.

exp

```
complex exp(const complex&)
```

Returns the complex value equal to *e* to the power of *x* where *x* is the argument.

log

```
complex log(complex)
```

Returns the natural logarithm of the argument *x*.

sinh

```
complex sinh(complex)
```

Returns the hyperbolic sine of the complex argument.

sinh

```
complex sinh(const complex&)
```

Returns the hyperbolic sine of the complex argument.

c_exception - Inherited Member Functions and Data

Inherited Public Functions

None

Inherited Public Data

None

Inherited Protected Functions

None

Inherited Protected Data

None

filebuf

The filebuf class specializes streambuf for using files as the ultimate producer of the ultimate consumer.

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

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

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

Class header file: fstream.h

filebuf - Hierarchy List

streambuf

filebuf

filebuf - Member Functions and Data by Group

Constructors & Destructor

You can construct and destruct objects of the filebuf class.

~filebuf

```
public:~filebuf()
```

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

filebuf

Overload 1

```
public:filebuf(int fd, char* p, long l)
```

Constructs a filebuf object that is attached to the file descriptor *fd*. The object is initialized to use the stream buffer starting at the position pointed to by *p* with length equal to *l*.

AIX Considerations

This function is available for 64-bit applications. The third argument is a long value.

Overload 2

```
public:filebuf(int fd)
```

Constructs a filebuf object that is attached to the file descriptor *fd*.

Overload 3

```
public:filebuf(int fd, char* p, int l)
```

Constructs a filebuf object that is attached to the file descriptor *fd*. The object is initialized to use the stream buffer starting at the position pointed to by *p* with length equal to *l*.

AIX Considerations

This function is available for 32-bit applications. The third argument is an int value.

Overload 4

```
public:filebuf()
```

Constructs an initially closed filebuf object.

Attach Functions

attach

Attaches the filebuf object to the file descriptor or the file pointer.

Overload 1

```
public:filebuf* attach(int fd)
```

Attaches the filebuf object to the file descriptor *fd*. If the filebuf object is already open or if *fd* is not open, attach() returns NULL. Otherwise, attach() returns a pointer to the filebuf object.

is_open

```
public:int is_open()
```

Returns a nonzero value if the filebuf object is attached to a file descriptor. Otherwise, is_open() returns zero.

open

```
public:filebuf*
  open( const char* name,
        int om,
        int prot = openprot )
```

Opens the file with the name name and attaches the filebuf object to it. If name does not already exist and the open mode om does not equal ios::nocreate, open() tries to create it with protection mode equal to prot. The default value of prot is filebuf::openprot. An error occurs if the filebuf

object is already open. If an error occurs, open() returns 0. Otherwise, open() returns a pointer to the filebuf object.

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

Data members

openprot

```
public:static const int openprot
```

The default protection mode used when opening files.

in_start

```
protected:char* in_start
```

Data member.

lahead

```
protected:char lahead [ 2 ]
```

A variable used to store look-ahead characters during underflow processing.

last_seek

```
protected:streampos last_seek
```

Stream position last seeked to.

mode

```
protected:int mode
```

Open mode of the filebuf object.

opened

```
protected:char opened
```

A flag used to track whether the file is open. If the file is open, the value of this variable is 1. Otherwise it is 0.

xfd

```
protected:int xfd
```

The file descriptor of the file attached to the filebuf object.

Detach Functions

close

```
public:filebuf* close()
```

close() does the following:

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

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

detach

```
public:int detach()
```

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

File Pointer Functions

overflow

```
public:virtual int overflow(int = EOF)
```

Emptys an output buffer. Returns EOF when an error occurs. Returns 0 otherwise.

seekoff

```
public:virtual streampos seekoff(streamoff, ios::seek_dir, int)
```

Moves the file get/put pointer to the position specified by the ios::seek_dir argument with the offset specified by the streamoff argument. ios::seek_dir can have the following values:

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

seekoff() changes the position of the file get/put pointer to the position specified by the value ios::seek_dir + streamoff. The offset can be either positive or negative. seekoff() ignores the third argument.

If the filebuf object is attached to a file that does not support seeking, or if the value of ios::seek_dir + streamoff specifies a position before the beginning of the file, seekoff() returns EOF and the position of the file get/put pointer is undefined. Otherwise, seekoff() returns the new position of the file get/put pointer.

sync

```
public:virtual int sync()
```

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

underflow

```
public:virtual int underflow()
```

Fills an input buffer. Returns EOF when an error occurs or the end of the input is reached. Returns the next character otherwise.

Query Functions

fd

```
public:int fd()
```

Returns the file descriptor that is attached to the filebuf object. If the filebuf object is closed, fd() returns EOF.

last_op

```
protected:int last_op()
```

Indicates whether the last operation was a read(get) or a write(put) operation.

Stream Buffer Functions

setbuf

Overload 1

```
public:virtual streambuf* setbuf(char* p, long len)
```

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

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

AIX Considerations

This function is available for 64-bit applications. The second argument is a long value

Overload 2

```
public:virtual streambuf* setbuf(char* p, int len)
```

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

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

AIX Considerations

This function is available for 32-bit applications. The second argument is an int value

filebuf - Inherited Member Functions and Data

Inherited Public Functions

streambuf	
Definition	**Page Number**
`virtual ~streambuf()`	235
`void dbp()`	238
`int in_avail()`	236

<table>
<tr><th colspan="2">streambuf</th></tr>
<tr><th>Definition</th><th>Page Number</th></tr>
<tr><td>long in_avail()</td><td>236</td></tr>
<tr><td>int optim_in_avail()</td><td>236</td></tr>
<tr><td>int optim_sbumpc()</td><td>236</td></tr>
<tr><td>int out_waiting()</td><td>242</td></tr>
<tr><td>long out_waiting()</td><td>242</td></tr>
<tr><td>virtual int overflow(int c = EOF)</td><td>242</td></tr>
<tr><td>virtual int pbackfail(int c)</td><td>243</td></tr>
<tr><td>int pptr_non_null()</td><td>239</td></tr>
<tr><td>int sbumpc()</td><td>237</td></tr>
<tr><td>virtual streampos
  seekoff(streamoff,
           ios::seek_dir,
           int = ios::in|ios::out)</td><td>239</td></tr>
<tr><td>virtual streampos
  seekpos(streampos,
           int = ios::in|ios::out)</td><td>239</td></tr>
<tr><td>streambuf* setbuf(unsigned char* p, int len)</td><td>244</td></tr>
<tr><td>streambuf* setbuf(char* p, int len, int count)</td><td>244</td></tr>
<tr><td>streambuf* setbuf(unsigned char* p, long len)</td><td>244</td></tr>
<tr><td>int sgetc()</td><td>237</td></tr>
<tr><td>long sgetn(char* s, long n)</td><td>237</td></tr>
<tr><td>int sgetn(char* s, int n)</td><td>237</td></tr>
<tr><td>int snextc()</td><td>237</td></tr>
<tr><td>int sputbackc(char c)</td><td>243</td></tr>
<tr><td>int sputc(int c)</td><td>243</td></tr>
<tr><td>long sputn(const char* s, long n)</td><td>243</td></tr>
<tr><td>int sputn(const char* s, int n)</td><td>243</td></tr>
<tr><td>void stossc()</td><td>240</td></tr>
<tr><td>streambuf(char* p, int l)</td><td>235</td></tr>
<tr><td>streambuf()</td><td>235</td></tr>
<tr><td>streambuf(char* p, int l, int c)</td><td>235</td></tr>
<tr><td>streambuf(char* p, long l)</td><td>235</td></tr>
<tr><td>virtual int xsgetn(char* s, int n)</td><td>238</td></tr>
<tr><td>virtual long xsgetn(char* s, long n)</td><td>238</td></tr>
<tr><td>virtual int xsputn(const char* s, int n)</td><td>244</td></tr>
</table>

streambuf	
Definition	**Page Number**
virtual long xsputn(const char* s, long n)	244

Inherited Public Data

None

Inherited Protected Functions

streambuf	
Definition	**Page Number**
int allocate()	246
char* base()	240
long blen() const	246
int blen() const	246
virtual int doallocate()	247
char* eback()	240
char* ebuf()	240
char* egptr()	240
char* epptr()	240
void gbump(long n)	240
void gbump(int n)	240
char* gptr()	241
char* pbase()	241
void pbump(int n)	241
void pbump(long n)	241
char* pptr()	241
void setb(char* b, char* eb, int a = 0)	242
void setg(char* eb, char* g, char* eg)	242
void setp(char* p, char* ep)	242
int unbuffered() const	247
void unbuffered(int unb)	247

Inherited Protected Data

None

fstream

This class specializes the iostream class for use with files.

Class header file: fstream.h

fstream - Hierarchy List

ios

fstreambase

fstream

fstream - Member Functions and Data by Group

Constructors & Destructor

Objects of the fstream class can be constructed and destructed.

~fstream

```
public:~fstream()
```

Destructs an fstream object.

fstream

Constructs an object of this class.

Overload 1

```
public:fstream(int fd, char* p, int l)
```

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

AIX Considerations

This function is available for 32-bit applications. The third argument is an int value.

Overload 2

```
public:fstream(int fd)
```

Constructs an fstream object that is attached to the file descriptor *fd*. If *fd* is not open, ios::failbit is set in the format state of the fstream object.

Overload 3

```
public:fstream(int fd, char* p, long l)
```

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

AIX Considerations

This function is available for 64-bit applications. The third argument is a long value.

Overload 4

```
public:fstream( const char* name,
        int mode,
        int prot = filebuf::openprot )
```

Constructs an fstream object and opens the file *name* with open mode equal to *mode* and protection mode equal to *prot*.

The default value for the argument *prot* is filebuf::openprot. If the file cannot be opened, the error state of the constructed fstream object is set.

Overload 5

```
public:fstream()
```

Constructs an unopened fstream object.

Filebuf Functions

Use these functions to work with the underlying filebuf object.

rdbuf

```
public:filebuf* rdbuf()
```

Returns a pointer to the filebuf object that is attached to the fstream object.

fstream - Inherited Member Functions and Data

Inherited Public Functions

fstreambase	
Definition	**Page Number**
`~fstreambase()`	133
`void attach(int fd)`	134
`void attach(FILE* fp)`	134
`void close()`	134
`int detach()`	134
`fstreambase( const char* name, int mode, int prot = filebuf::openprot )`	133
`fstreambase(int fd, char* p, long l)`	133
`fstreambase()`	133
`fstreambase(int fd)`	133
`fstreambase( const char* name, const char* attr, int mode, int prot = filebuf::openprot )`	133
`fstreambase(int fd, char* p, int l)`	133
`void setbuf(char* p, int l)`	135
`void setbuf(char* p, long l)`	135

ios	
Definition	**Page Number**
virtual ~ios()	143
int bad() const	144
static long bitalloc()	149
void clear(int i = 0)	145
int eof() const	145
int fail() const	145
char fill(char)	145
char fill() const	145
long flags(long f)	146
long flags() const	146
int good() const	145
ios(streambuf*)	143
long& iword(int)	149
int operator !() const	149
operator const void *() const	149
operator void *()	149
int precision() const	146
int precision(int)	146
void *& pword(int)	149
streambuf* rdbuf()	149
int rdstate() const	145
long setf(long setbits, long field)	147
long setf(long)	147
int skip(int i)	147
static void sync_with_stdio()	149
ostream* tie()	150
ostream* tie(ostream* s)	150
long unsetf(long)	147
int width() const	148
int width(int w)	148
static int xalloc()	151
static long xalloc()	151

Inherited Public Data

ios	
Definition	**Page Number**
static const long adjustfield	143
static const long basefield	144
static const long floatfield	144

Inherited Protected Functions

ios	
Definition	**Page Number**
void init(streambuf*)	149
ios()	143
void setstate(int b)	145

fstreambase	
Definition	**Page Number**
void verify(int)	135

Inherited Protected Data

ios	
Definition	**Page Number**
static void (* stdioflush) ( )	151
int assign_private	144
streambuf* bp	144
int delbuf	144
int isfx_special	144
int ispecial	144
int osfx_special	144
int ospecial	144
int state	144
char x_fill	148
long x_flags	144
short x_precision	148
ostream* x_tie	144
short x_width	148

fstreambase

The fstreambase class is an internal class that provides common functions for the classes that are derived from it; fstream, ifstream and ofstream. The fstreambase class inherits from the ios class. Do not use the fstreambase class directly.

Class header file: fstream.h

fstreambase - Hierarchy List

ios
fstreambase
ifstream
fstream
ofstream

fstreambase - Member Functions and Data by Group

Constructors & Destructor

Objects of the fstreambase class can be constructed and destructed by objects that derive from it. These constructors and destructors should not be used directly.

~fstreambase

```
public:~fstreambase()
```

Destructs an fstreambase object.

fstreambase

Constructs an object of this class.

Overload 1

```
public:fstreambase(int fd, char* p, int l)
```

This constructor does the following:

- constructs an fstreambase object
- initializes the filebuf object to the file descriptor passed in
- initializes the streambuf object and sets the get and put pointers based on the pointer **p** and the length **l**
- initializes the ios object.

If the file is already open, it clears the ios state. Otherwise, it sets the ios::failbit in the format state of the object.

AIX Considerations

This function is available for 32-bit applications. The third argument is an int value.

Overload 2

```
public:fstreambase(int fd)
```

This constructor does the following:

- constructs an fstreambase object
- initializes the filebuf object to the file descriptor passed in
- initializes the ios object.

If the file is already open, it clears the ios state. Otherwise, it sets the ios::failbit in the format state of the object.

Overload 3

```
public:fstreambase(int fd, char* p, long l)
```

This constructor does the following:

- constructs an fstreambase object
- initializes the filebuf object to the file descriptor passed in
- initializes the streambuf object and sets the get and put pointers based on the pointer **p** and the length **l**
- initializes the ios object.

If the file is already open, it clears the ios state. Otherwise, it sets the ios::failbit in the format state of the object.

AIX Considerations

This function is available for 64-bit applications. The third argument is a long value.

Overload 4

```
public:fstreambase( const char* name,
          int mode,
          int prot = filebuf::openprot )
```

Constructs an fstreambase object, initializes the ios object, and opens the specified file with the specified mode and protection.

Overload 5

```
public:fstreambase()
```

Default constructor. Constructs an fstreambase object and initializes the ios object.

Filebuf Functions

Use these functions to work with the underlying filebuf object.

attach

Attaches the fstream, ifstream or ofstream object to the file descriptor or file pointer.

Overload 1

```
public:void attach(int fd)
```

Attaches the fstream, ifstream or ofstream object to the file descriptor *fd*. If the object is already attached to a file descriptor, an error occurs and ios::failbit is set in the format state of the object.

close

```
public:void close()
```

Closes the filebuf object, breaking the connection between the fstream, ifstream or ofstream object and the file descriptor. close() calls *filebuf*->close(). If this call fails, the error state of the fstream, ifstream or ofstream object is not cleared.

detach

```
public:int detach()
```

Detaches the filebuf object, breaking the connection between the fstream, ifstream or ofstream object and the file descriptor. detach() calls *filebuf*->detach().

rdbuf

```
public:filebuf* rdbuf()
```

Returns a pointer to the filebuf object that is attached to the fstream, ifstream or ofstream object.

Miscellaneous Functions

verify

```
protected:void verify(int)
```

Clears the format state of the object or sets the ios::failbit in the format state of the object depending on the value of the argument. If the argument value is 1, the format state is cleared, otherwise the ios::failbit is set.

Open Functions

open

Opens the specified file.

Overload 1

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

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

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

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

Stream Buffer Functions

Use these functions to work with the underlying streambuf object.

setbuf

Overload 1

```
public:void setbuf(char* p, long l)
```

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

AIX Considerations

This function is available for 64-bit applications. The second argument is a long value.

Overload 2

```
public:void setbuf(char* p, int l)
```

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

AIX Considerations

This function is available for 32-bit applications. The second argument is an int value.

fstreambase - Inherited Member Functions and Data

Inherited Public Functions

ios	
Definition	**Page Number**
`virtual ~ios()`	143
`int bad() const`	144
`static long bitalloc()`	149
`void clear(int i = 0)`	145
`int eof() const`	145
`int fail() const`	145
`char fill(char)`	145
`char fill() const`	145
`long flags(long f)`	146
`long flags() const`	146
`int good() const`	145
`ios(streambuf*)`	143
`long& iword(int)`	149
`int operator !() const`	149
`operator const void *() const`	149
`operator void *()`	149
`int precision() const`	146
`int precision(int)`	146
`void *& pword(int)`	149
`streambuf* rdbuf()`	149
`int rdstate() const`	145
`long setf(long setbits, long field)`	147
`long setf(long)`	147

ios	
Definition	**Page Number**
int skip(int i)	147
static void sync_with_stdio()	149
ostream* tie(ostream* s)	150
ostream* tie()	150
long unsetf(long)	147
int width(int w)	148
int width() const	148
static int xalloc()	151
static long xalloc()	151

Inherited Public Data

ios	
Definition	**Page Number**
static const long adjustfield	143
static const long basefield	144
static const long floatfield	144

Inherited Protected Functions

ios	
Definition	**Page Number**
void init(streambuf*)	149
ios()	143
void setstate(int b)	145

Inherited Protected Data

ios	
Definition	**Page Number**
static void (* stdioflush) ( )	151
int assign_private	144
streambuf* bp	144
int delbuf	144
int isfx_special	144
int ispecial	144

ios	
Definition	**Page Number**
int osfx_special	144
int ospecial	144
int state	144
char x_fill	148
long x_flags	144
short x_precision	148
ostream* x_tie	144
short x_width	148

ifstream

This class specializes the istream class for use with files.

Class header file: fstream.h

ifstream - Hierarchy List

ios
fstreambase
ifstream

ifstream - Member Functions and Data by Group

Constructors & Destructor

Objects of the ifstream class can be constructed and destructed.

~ifstream

```
public:~ifstream()
```

Destructs an ifstream object.

ifstream

Constructs an object of this class.

Overload 1

```
public:ifstream(int fd, char* p, int l)
```

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

AIX Considerations

This function is available for 32-bit applications. The third argument is an int value.

Overload 2

```
public:ifstream(int fd)
```

Constructs an ifstream object that is attached to the file descriptor *fd*. If *fd* is not open, ios::failbit is set in the format state of the ifstream object.

Overload 3

```
public:ifstream(int fd, char* p, long l)
```

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

AIX Considerations

This function is available for 64-bit applications. The third argument is a long value.

Overload 4

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

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

Overload 5

```
public:ifstream()
```

Constructs an unopened ifstream object.

Filebuf Functions

rdbuf

```
public:filebuf* rdbuf()
```

Returns a pointer to the filebuf object that is attached to the ifstream object.

Open Functions

Opens the file.

open

Opens the specified file.

Overload 1

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

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

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

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

ifstream - Inherited Member Functions and Data

Inherited Public Functions

fstreambase	
Definition	**Page Number**
`~fstreambase()`	133
`void attach(FILE* fp)`	134
`void attach(int fd)`	134
`void close()`	134
`int detach()`	134

`fstreambase( const char* name,`
`int mode,`
`int prot = filebuf::openprot )`
	133

`fstreambase()`	133

`fstreambase( const char* name,`
`const char* attr,`
`int mode,`
`int prot = filebuf::openprot )`
	133

`fstreambase(int fd, char* p, int l)`	133
`fstreambase(int fd, char* p, long l)`	133
`fstreambase(int fd)`	133

`void`
`open( const char* name,`
`int mode,`
`int prot = filebuf::openprot )`
	135

`void`
`open( const char* name,`
`const char* attr,`
`int mode,`
`int prot = filebuf::openprot )`
	135

`void setbuf(char* p, int l)`	135
`void setbuf(char* p, long l)`	135

ios	
Definition	**Page Number**
`virtual ~ios()`	143

ios	
Definition	**Page Number**
int bad() const	144
static long bitalloc()	149
void clear(int i = 0)	145
int eof() const	145
int fail() const	145
char fill(char)	145
char fill() const	145
long flags() const	146
long flags(long f)	146
int good() const	145
ios(streambuf*)	143
long& iword(int)	149
int operator !() const	149
operator const void *() const	149
operator void *()	149
int precision() const	146
int precision(int)	146
void *& pword(int)	149
streambuf* rdbuf()	149
int rdstate() const	145
long setf(long setbits, long field)	147
long setf(long)	147
int skip(int i)	147
static void sync_with_stdio()	149
ostream* tie()	150
ostream* tie(ostream* s)	150
long unsetf(long)	147
int width(int w)	148
int width() const	148
static int xalloc()	151
static long xalloc()	151

Inherited Public Data

ios	
Definition	**Page Number**
static const long adjustfield	143
static const long basefield	144
static const long floatfield	144

Inherited Protected Functions

ios	
Definition	**Page Number**
void init(streambuf*)	149
ios()	143
void setstate(int b)	145

fstreambase	
Definition	**Page Number**
void verify(int)	135

Inherited Protected Data

ios	
Definition	**Page Number**
static void (* stdioflush) ( )	151
int assign_private	144
streambuf* bp	144
int delbuf	144
int isfx_special	144
int ispecial	144
int osfx_special	144
int ospecial	144
int state	144
char x_fill	148
long x_flags	144
short x_precision	148
ostream* x_tie	144
short x_width	148

ios

The ios class is the base class for the classes that format data that is extracted from or inserted into the stream buffer. The derived classes support the movement of formatted and unformatted data to and from the stream buffer.

The ios class maintains the format and error state information for the classes that are derived from it. The format state is a collection of flags and variables that can be set to control the details of formatting operations for input and output. The error state is a collection of flags that records whether any errors have taken place in the processing of the ios object. It also recoreds whether the end of an input stream has been reached.

Class header file: iostream.h

ios - Hierarchy List

ios

ostream

fstreambase

stdiostream

strstreambase

istream

ios - Member Functions and Data by Group

Constructors & Destructor

Objects of the ios class can be constructed and destructed.

~ios

```
public:virtual ~ios()
```

Destructs an ios object.

ios

Creates an ios object.

Overload 1

```
public:ios(streambuf*)
```

The streambuf object is associated with the constructed ios object. If this argument is equal to 0, the result is undefined.

Overload 2

```
protected:ios()
```

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

Data members

adjustfield

```
public:static const long adjustfield
```

Data member for the ios class.

basefield

```
public:static const long basefield
```

Data member for the ios class.

floatfield

```
public:static const long floatfield
```

Data member for the ios class.

assign_private

```
protected:int assign_private
```

Data member for the ios class.

bp

```
protected:streambuf* bp
```

Data member for the ios class. Pointer to the streambuf object.

delbuf

```
protected:int delbuf
```

Data member for the ios class.

isfx_special

```
protected:int isfx_special
```

Data member for the ios class.

ispecial

```
protected:int ispecial
```

Data member for the ios class.

osfx_special

```
protected:int osfx_special
```

Data member for the ios class.

ospecial

```
protected:int ospecial
```

Data member for the ios class.

state

```
protected:int state
```

Data member for the ios class.

x_flags

```
protected:long x_flags
```

Data member for the ios class.

x_tie

```
protected:ostream* x_tie
```

Data member for the ios class.

Error State Functions

bad

```
public:int bad() const
```

Returns a nonzero value is ios::badbit is set in the error state of the ios object. Otherwise, it returns 0.

ios::badbit is usually set when some operation on the streambuf object that is associated with the ios object has failed. It will probably not be possible to continue input and output operations on the ios object.

clear

```
public:void clear(int i = 0)
```

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

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

eof

```
public:int eof() const
```

Returns a nonzero value if ios::eofbit is set in the error state of the ios object. Otherwise, it returns 0.

ios::eofbit is usually set when an EOF has been encountered during an extraction operation.

fail

```
public:int fail() const
```

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

good

```
public:int good() const
```

Returns a nonzero value if no bits are set in the error state of the ios object. Otherwise, it returns 0.

rdstate

```
public:int rdstate() const
```

Returns the current value of the error state of the ios object.

setstate

```
protected:void setstate(int b)
```

Format State Functions

fill

Overload 1

```
public:char fill() const
```

Returns the value of ios::x_fill of the ios object.

ios::x_fill is the character used as padding if the field is wider than the representation of a value. The default value for ios::x_fill is a space. The ios::left, ios::right and ios::internal flags determine the position of the fill character.

You can also use the parameterized manipulator *setfill* to set the value of ios::x_fill.

Overload 2

```
public:char fill(char)
```

Sets the value of ios::x_fill of the ios object to the specified character.

ios::x_fill is the character used as padding if the field is wider than the representation of a value. The default value for ios::x_fill is a space. The ios::left, ios::right and ios::internal flags determine the position of the fill character.

You can also use the parameterized manipulator *setfill* to set the value of ios::x_fill.

flags

Overload 1

```
public:long flags() const
```

Returns the value of the flags that make up the current format state.

Overload 2

```
public:long flags(long f)
```

Sets the flags in the format state to the settings specified in the argument and returns the value of the previous settings of the format flags.

precision

Overload 1

```
public:int precision() const
```

Returns the value of ios::x_precision.

ios::x_precision controls the number of significant digits when floating-point values are inserted.

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

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

Overload 2

```
public:int precision(int)
```

Sets the value of ios::x_precision to the specified value and returns the previous value. The value must be greater than 0. If the value is negative, the value of ios::x_precision is set to the default value, 6.

You can also use the parameterized manipulator *setprecision* to set ios::x_precision.

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

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

setf

Overload 1

```
public:long setf(long setbits, long field)
```

This function clears the format flags specified in *field*, sets the format flags specified in *setbits*, and returns the previous value of the format state.

For example, to change the conversion base in the format state to ios::hex, you could use a statement like this:

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

In this statement, ios::basefield specifies the conversion base as the format flag that is going to be changed and ios::hex specifies the new value for the conversion base. If *setbits* equals 0, all of the format flags specified in *field* are cleared.

You can also use the parameterized manipulator *resetiosflags* to clear format flags.

Note: If you set conflicting flags the results are unpredictable.

Overload 2

```
public:long setf(long)
```

This function is accumulative. It sets the format flags that are specified in the argument, without affecting format flags that are not marked in the argument, and returns the previous value of the format state.

You can also use the parameterized manipulator *setiosflags* to set the format flags to a specific setting.

skip

```
public:int skip(int i)
```

Sets the format flag ios::skipws if the value of the argument *i* does not equal 0. If *i* does equal 0, ios::skipws is cleared.

skip() returns a value of 1 if ios::skipws was set prior to the call to skip(), and returns 0 otherwise.

unsetf

```
public:long unsetf(long)
```

Turns off the format flags specified in the argument and returns the previous format state.

width

Overload 1

```
public:int width() const
```

Returns the value of the current setting of the format state field width variable, ios::x_width.

If the value of ios::x_width is smaller than the space needed for the representation of the value, the full value is still inserted.

Overload 2

```
public:int width(int w)
```

Sets ios::x_width to the value *w* and returns the previous value.

The default field width is 0. When the value of ios::x_width is 0, the operations that insert values only insert the characters needed to represent a value.

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

You can also use the parameterized manipulator *setw* to set the field width.

Format State Variables

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

x_fill

```
protected:char x_fill
```

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

x_precision

```
protected:short x_precision
```

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

x_width

```
protected:short x_width
```

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

Initialization Functions

init

```
protected:void init(streambuf*)
```

Miscellaneous Functions

bitalloc

```
public:static long bitalloc()
```

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

iword

```
public:long& iword(int)
```

Returns a reference to the indexed user-defined flag, where the index used in the argument to this function is returned by xalloc().

iword() allocates space for the user-defined flag. If the allocation fails, iword() sets ios::failbit. You should check ios::failbit after calling iword().

operator !

```
public:int operator !() const
```

The ! operator returns a nonzero value if ios::failbit or ios::badbit is set in the error state of the ios object.

For example, you could write:

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

operator const void *

```
public:operator const void *() const
```

operator void *

```
public:operator void *()
```

pword

```
public:void *& pword(int)
```

Returns a reference to a pointer to the indexed user-defined flag where the index used in the argument to this function is returned by xalloc().

pword() allocates space for the user-defined flag. If the allocation fails, pword() sets ios::failbit. You should check ios::failbit after calling pword().

pword() is the same as iword(), except that the two functions return different types.

rdbuf

```
public:streambuf* rdbuf()
```

Returns a pointer to the streambuf object that is associated with the ios object. This is the streambuf object that was passed as an argument to the ios constructor.

sync_with_stdio

```
public:static void sync_with_stdio()
```

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

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

sync_with_stdio() makes cout and clog *unit buffered*. After you call sync_with_stdio(), the performance of your program could diminish. The performance of your program depends on the length of strings, with performance diminishing most when the strings are shortest.

tie

Overload 1

```
public:ostream* tie()
```

Returns the value of ios::x_tie.

ios::x_tie is the tie variable that points to the ostream object that is tied to the ios object.

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

By default, the tie variables of the predefined streams cin, cerr and clog all point to the predefined stream cout.

Overload 2

```
public:ostream* tie(ostream* s)
```

Sets the tie variable, ios::x_tie, equal to the specified ostream and returns the previous value.

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

By default, the tie variables of the predefined streams cin, cerr and clog all point to the predefined stream cout.

xalloc

A static function that returns an unused index into an array of words available for use as format state variables by classes derived from ios.

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

Overload 1

```
public:static int xalloc()
```

AIX Considerations

The value returned is an int for 32-bit applications. This function is not available for 64-bit applications.

Overload 2

```
public:static long xalloc()
```

AIX Considerations

The value returned is a long for 64-bit applications. This function is not available for 32-bit applications.

(* stdioflush) ( )

```
protected:static void ( * stdioflush ) ( )
```

ios - Enumerations

Variation 1

```
enum  { skipping=01000,
        tied=02000 }
```

Variation 2

```
enum  { skipws=01,
        left=02,
        right=04,
        internal=010,
        dec=020,
        oct=040,
        hex=0100,
        showbase=0200,
        showpoint=0400,
        uppercase=01000,
        showpos=02000,
        scientific=04000,
        fixed=010000,
        unitbuf=020000,
        stdio=040000 }
```

io_state

The error state state is an enumeration that records the errors that take place in the processing of ios objects.

Note: hardfail is a flag used internally by the I/O Stream Library. Do not use it.

open_mode

The elements of the open_mode enumeration have the following meanings:

- ios::app - open() performs a seek to the end of the file. Data that is written is appended to the end of the file. This value implies that the file is open for output.

- ios::ate - open() performs a seek to the end of the file. Setting ios::ate does not open the file for input or output. If you set ios::ate, you should explicitly set ios::in, ios::out, or both.
- ios::bin - See ios::binary below.
- ios::binary - The file is opened in binary mode. In the default (text) mode, carriage returns are discarded on input, as in an end-of-file (0x1a) character if it is the last character in the file. This means that a carriage return without an accompanying line feed causes the characters on either side of the carriage return to become adjacent. On output, a line feed is expanded to a carriage return and line feed. If you specify ios::binary, carriage returns and terminating end-of-file characters are not removed on input, and a line feed is not expanded to a carriage return and line feed on output. ios::binary and ios::bin provide identical functionality.
- ios::in - The file is opened for input. If the file that is being opened for input does not exist, the open operation will fail. ios::noreplace is ignoredif ios::in is set.
- ios::out - The file is opened for output.
- ios::trunc - If the file already exists, its contents will be discarded. If you specify ios::out and neither ios::ate nor ios::app, you are implicitly specifying ios::trunc. If you set ios::trunc, you should explicitly set ios::in, ios::out, or both.
- ios::nocreate - If the file does not exist, the call to open() fails.
- ios::noreplace - If the file already exists and ios::out is et, the call to open() fails. If ios::out is not set, ios::noreplace is ignored.

Variation 1

```
enum open_mode { in=1,
                 out=2,
                 ate=4,
                 app=010,
                 trunc=020,
                 nocreate=040,
                 noreplace=0100,
                 bin=0200,
                 binary=bin }
```

seek_dir

The elements of the seek_dir enumeration have the following meanings:

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

ios - Inherited Member Functions and Data

Inherited Public Functions

None

Inherited Public Data

None

Inherited Protected Functions

None

Inherited Protected Data

None

iostream

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

- iostream_withassign - to assign another stream (such as an fstream for a file) to an iostream object.
- strstream - a stream of characters stored in memory.
- fstream - a stream that supports input and output.

Class header file: iostream.h

iostream - Hierarchy List

ios

istream

iostream

iostream_withassign

iostream - Member Functions and Data by Group

Constructors & Destructor

Objects of the iostream class can be constructed and destructed.

~iostream

```
public:virtual ~iostream()
```

Destructs an iostream object.

iostream

Overload 1

```
public:iostream(streambuf*)
```

This constructor takes a single streambuf argument and creates an iostream object that is attached to the streambuf object. The constructor also initializes the format variables to their defaults.

Overload 2

```
protected:iostream()
```

Protected contructor.

iostream - Inherited Member Functions and Data

Inherited Public Functions

ios	
Definition	**Page Number**
`virtual ~ios()`	143
`int bad() const`	144
`static long bitalloc()`	149
`void clear(int i = 0)`	145
`int eof() const`	145

ios	
Definition	**Page Number**
int fail() const	145
char fill() const	145
char fill(char)	145
long flags(long f)	146
long flags() const	146
int good() const	145
ios(streambuf*)	143
long& iword(int)	149
int operator !() const	149
operator const void *() const	149
operator void *()	149
int precision() const	146
int precision(int)	146
void *& pword(int)	149
streambuf* rdbuf()	149
int rdstate() const	145
long setf(long)	147
long setf(long setbits, long field)	147
int skip(int i)	147
static void sync_with_stdio()	149
ostream* tie(ostream* s)	150
ostream* tie()	150
long unsetf(long)	147
int width(int w)	148
int width() const	148
static long xalloc()	151
static int xalloc()	151

istream	
Definition	**Page Number**
virtual ~istream()	165
long gcount()	165
int gcount()	165

istream	
Definition	**Page Number**
istream& get(char& c)	166
int get()	166
istream& get(signed char* b, long lim, char delim = '\n')	166
istream& get(signed char* b, int lim, char delim = '\n')	166
istream& get(char*, long lim, char delim = '\n')	166
istream& get(signed char& c)	166
istream& get(unsigned char& c)	166
istream& get(unsigned char* b, long lim, char delim = '\n')	166
istream& get(char*, int lim, char delim = '\n')	166
istream& get(streambuf& sb, char delim = '\n')	166
istream& get(unsigned char* b, int lim, char delim = '\n')	166
istream& get(wchar_t&)	166
istream& get_complicated(char& c)	169
istream& get_complicated(unsigned char& c)	169
istream& get_complicated(signed char& c)	169
istream& getline(unsigned char* b, long lim, char delim = '\n')	169
istream& getline(unsigned char* b, int lim, char delim = '\n')	169
istream& getline(signed char* b, int lim, char delim = '\n')	169
istream& getline(char* b, long lim, char delim = '\n')	169
istream& getline(char* b, int lim, char delim = '\n')	169
istream& getline(signed char* b, long lim, char delim = '\n')	169
istream& ignore(int n = 1, int delim = EOF)	172
int ipfx(int noskipws = 0)	185
int ipfx(long noskipws = 0)	185
void isfx()	186
istream(int fd, int sk = 1, ostream* t = 0)	165

istream	
Definition	**Page Number**
istream(streambuf*, int sk, ostream* t = 0)	165
istream(int size, char*, int sk = 1)	165
istream(streambuf*)	165
istream& operator >>(istream & (* f) (istream &))	174
istream& operator >>(char*)	174
istream& operator >>(unsigned int&)	174
istream& operator >>(long double&)	174
istream& operator >>(long long&)	174
istream& operator >>(signed char& c)	174
istream& operator >>(ios & (* f) (ios &))	174
istream& operator >>(double&)	174
istream& operator >>(streambuf*)	174
istream& operator >>(char& c)	174
istream& operator >>(signed char*)	174
istream& operator >>(unsigned long&)	174
istream& operator >>(short&)	174
istream& operator >>(unsigned char*)	174
istream& operator >>(float&)	174
istream& operator >>(wchar_t&)	174
istream& operator >>(int&)	174
istream& operator >>(long&)	174
istream& operator >>(unsigned short&)	174
istream& operator >>(unsigned long long&)	174
istream& operator >>(unsigned char& c)	174
istream& operator >>(wchar_t*)	174
int peek()	172
istream& putback(char c)	184
istream& read(signed char* s, long n)	172
istream& read(char* s, int n)	172
istream& read(unsigned char* s, int n)	172
istream& read(signed char* s, int n)	172
istream& read(unsigned char* s, long n)	172
istream& read(char* s, long n)	172

istream	
Definition	**Page Number**
`istream& rs_complicated(unsigned char& c)`	174
`istream& rs_complicated(signed char& c)`	174
`istream& rs_complicated(char& c)`	174
`istream& seekg(streamoff o, ios::seek_dir d)`	184
`istream& seekg(streampos p)`	184
`int sync()`	185
`streampos tellg()`	185

Inherited Public Data

ios	
Definition	**Page Number**
`static const long adjustfield`	143
`static const long basefield`	144
`static const long floatfield`	144

Inherited Protected Functions

istream	
Definition	**Page Number**
`int do_ipfx(long noskipws)`	186
`int do_ipfx(int noskipws)`	186
`void eatwhite()`	174
`istream()`	165

ios	
Definition	**Page Number**
`void init(streambuf*)`	149
`ios()`	143
`void setstate(int b)`	145

Inherited Protected Data

ios	
Definition	**Page Number**
`static void ( * stdioflush ) ( )`	151

ios	
Definition	**Page Number**
int assign_private	144
streambuf* bp	144
int delbuf	144
int isfx_special	144
int ispecial	144
int osfx_special	144
int ospecial	144
int state	144
char x_fill	148
long x_flags	144
short x_precision	148
ostream* x_tie	144
short x_width	148

iostream_withassign

This class is derived from istream_withassign and ostream_withassign. Use this class to assign another stream to an iostream object.

Class header file: iostream.h

iostream_withassign - Hierarchy List

ios
istream
iostream
iostream_withassign

iostream_withassign - Member Functions and Data by Group

Constructors & Destructor

~iostream_withassign

```
public:virtual ~iostream_withassign()
```

Destructs an iostream_withassign object.

iostream_withassign

```
public:iostream_withassign()
```

Creates an iostream_withassign object. It does not do any initialization of this object.

operator =

```
public:iostream_withassign& operator =(iostream_withassign& rhs)
```

Copy constructor.

Assignment Operators

operator =

Overload 1

```
public:iostream_withassign& operator =(streambuf*)
```

This assignment operator takes a pointer to a streambuf object and associates this streambuf object with the iostream_withassign object that is on the left side of the assignment operator.

Overload 2

```
public:iostream_withassign& operator =(ios&)
```

This assignment operator takes a reference to an ios object and associates the stream buffer attached to this ios object with the iostream_withassign object that is on the left side of the assignment operator.

iostream_withassign - Inherited Member Functions and Data

Inherited Public Functions

ios	
Definition	**Page Number**
virtual ~ios()	143
int bad() const	144
static long bitalloc()	149
void clear(int i = 0)	145
int eof() const	145
int fail() const	145
char fill() const	145
char fill(char)	145
long flags() const	146
long flags(long f)	146
int good() const	145
ios(streambuf*)	143
long& iword(int)	149
int operator !() const	149
operator const void *() const	149
operator void *()	149
int precision() const	146
int precision(int)	146
void *& pword(int)	149

ios	
Definition	**Page Number**
`streambuf* rdbuf()`	149
`int rdstate() const`	145
`long setf(long)`	147
`long setf(long setbits, long field)`	147
`int skip(int i)`	147
`static void sync_with_stdio()`	149
`ostream* tie(ostream* s)`	150
`ostream* tie()`	150
`long unsetf(long)`	147
`int width() const`	148
`int width(int w)`	148
`static long xalloc()`	151
`static int xalloc()`	151

iostream	
Definition	**Page Number**
`virtual ~iostream()`	153
`iostream(streambuf*)`	153

istream	
Definition	**Page Number**
`virtual ~istream()`	165
`int gcount()`	165
`long gcount()`	165
`istream& get(unsigned char& c)`	166
`istream& get(char& c)`	166
`istream& get(char*, long lim, char delim = '\n')`	166
`istream& get(streambuf& sb, char delim = '\n')`	166
`istream& get(unsigned char* b, int lim, char delim = '\n')`	166
`int get()`	166
`istream& get(unsigned char* b, long lim, char delim = '\n')`	166
`istream& get(signed char* b, int lim, char delim = '\n')`	166

istream	
Definition	**Page Number**
`istream& get(signed char* b, long lim, char delim = '\n')`	166
`istream& get(char*, int lim, char delim = '\n')`	166
`istream& get(wchar_t&)`	166
`istream& get(signed char& c)`	166
`istream& get_complicated(unsigned char& c)`	169
`istream& get_complicated(char& c)`	169
`istream& get_complicated(signed char& c)`	169

`istream&`
`getline( unsigned char* b,`
`long lim,`
`char delim = '\n' )`
	169

`istream&`
`getline( signed char* b,`
`int lim,`
`char delim = '\n' )`
	169

`istream&`
`getline( unsigned char* b,`
`int lim,`
`char delim = '\n' )`
	169

`istream& getline(char* b, int lim, char delim = '\n')`	169
`istream& getline(char* b, long lim, char delim = '\n')`	169

`istream&`
`getline( signed char* b,`
`long lim,`
`char delim = '\n' )`
	169

`istream& ignore(int n = 1, int delim = EOF)`	172
`int ipfx(int noskipws = 0)`	185
`int ipfx(long noskipws = 0)`	185
`void isfx()`	186
`istream(int size, char*, int sk = 1)`	165
`istream(int fd, int sk = 1, ostream* t = 0)`	165
`istream(streambuf*, int sk, ostream* t = 0)`	165
`istream(streambuf*)`	165
`istream& operator >>(unsigned char& c)`	174
`istream& operator >>(istream & ( * f ) ( istream & ))`	174
`istream& operator >>(char*)`	174
`istream& operator >>(unsigned int&)`	174
`istream& operator >>(long long&)`	174

istream	
Definition	**Page Number**
istream& operator >>(long double&)	174
istream& operator >>(signed char& c)	174
istream& operator >>(int&)	174
istream& operator >>(ios & (* f) (ios &))	174
istream& operator >>(streambuf*)	174
istream& operator >>(double&)	174
istream& operator >>(wchar_t&)	174
istream& operator >>(char& c)	174
istream& operator >>(signed char*)	174
istream& operator >>(unsigned long&)	174
istream& operator >>(short&)	174
istream& operator >>(unsigned char*)	174
istream& operator >>(float&)	174
istream& operator >>(wchar_t*)	174
istream& operator >>(long&)	174
istream& operator >>(unsigned long long&)	174
istream& operator >>(unsigned short&)	174
int peek()	172
istream& putback(char c)	184
istream& read(signed char* s, long n)	172
istream& read(char* s, int n)	172
istream& read(unsigned char* s, int n)	172
istream& read(signed char* s, int n)	172
istream& read(unsigned char* s, long n)	172
istream& read(char* s, long n)	172
istream& rs_complicated(unsigned char& c)	174
istream& rs_complicated(signed char& c)	174
istream& rs_complicated(char& c)	174
istream& seekg(streamoff o, ios::seek_dir d)	184
istream& seekg(streampos p)	184
int sync()	185
streampos tellg()	185

Inherited Public Data

ios	
Definition	**Page Number**
static const long adjustfield	143
static const long basefield	144
static const long floatfield	144

Inherited Protected Functions

istream	
Definition	**Page Number**
int do_ipfx(long noskipws)	186
int do_ipfx(int noskipws)	186
void eatwhite()	174
istream()	165

ios	
Definition	**Page Number**
void init(streambuf*)	149
ios()	143
void setstate(int b)	145

iostream	
Definition	**Page Number**
iostream()	153

Inherited Protected Data

ios	
Definition	**Page Number**
static void (* stdioflush) ( )	151
int assign_private	144
streambuf* bp	144
int delbuf	144
int isfx_special	144
int ispecial	144
int osfx_special	144

ios	
Definition	**Page Number**
`int ospecial`	144
`int state`	144
`char x_fill`	148
`long x_flags`	144
`short x_precision`	148
`ostream* x_tie`	144
`short x_width`	148

istream

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

```
ins >> x;
```

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

The details of conversion depend on the format state of the istream object and the type of the variable *x*. The input operator may set the width variable ios::x_width to 0, but it does not change anything else in the format state.

The input operator is defined for the following types:
- Arrays of character values (including signed char and unsigned char)
- Other integral values: short, int, long, float, double, long double, and long long values.

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

You can also define input operators for your own types.

Class header file: iostream.h

istream - Hierarchy List

ios
istream
iostream
istream_withassign

istream - Member Functions and Data by Group

Constructors & Destructor

Objects of the istream class can be constructed and destructed.

~istream

```
public:virtual ~istream()
```

Destructs an istream object.

istream

Overload 1

```
public:istream(streambuf*, int sk, ostream* t = 0)
```

Obsolete. Do not use.

Overload 2

```
public:istream(streambuf*)
```

This constructor takes a single argument, a pointer to a streambuf, and creates an istream object that is attached to the streambuf object. The constructor also initializes the format variables to their defaults.

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

Overload 3

```
public:istream(int size, char*, int sk = 1)
```

Obsolete. Do not use.

Overload 4

```
public:istream(int fd, int sk = 1, ostream* t = 0)
```

Obsolete. Do not use.

Overload 5

```
protected:istream()
```

Obsolete. Do not use.

Extract Functions

You can use the extract functions to extract characters from a stream buffer as a sequence of bytes. All of these functions call ipfx(1). They only proceed with their processing if ipfx(1) returns a nonzero value.

gcount

Returns the number of characters extracted from the stream buffer by the last call to an unformatted input function. The input operator >> may call unformatted input functions, and thus formatted input may affect the value returned by gcount().

Overload 1

```
public:int gcount()
```

AIX Considerations

This function returns an int value for 32-bit applications. It is not available for 64-bit applications.

Overload 2

```
public:long gcount()
```

AIX Considerations

This function returns a long value for 64-bit applications. It is not available for 32-bit applications.

get

Overload 1

```
public:int get()
```

Extracts a single character from the stream buffer attached to the istream object and returns it. Returns EOF if EOF is extracted. ios::failbit is never set.

Overload 2

```
public:istream& get(char*, int lim, char delim = '\n')
```

Extracts characters from the stream buffer attached to the istream object and stores them in the byte array beginning at the location pointed to by the first argument and extending for *lim* bytes. The default value of the *delim* argument is '\n'. Extraction stops when either of the following conditions is true:

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

get() always stores a terminating null character in the array, even if it does not extract any characters from the stream buffer. ios::failbit is set if EOF is encountered before any characters are stored.

AIX Considerations

This function is available for 32-bit applications. The second argument is an int value.

Overload 3

```
public:istream& get(unsigned char& c)
```

Extracts a single character from the stream buffer attached to the istream object and stores this character in *c*.

Overload 4

```
public:istream& get(signed char* b, int lim, char delim = '\n')
```

Extracts characters from the stream buffer attached to the istream object and stores them in the byte array beginning at the location pointed to by the first argument and extending for *lim* bytes. The default value of the *delim* argument is '\n'. Extraction stops when either of the following conditions is true:

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

get() always stores a terminating null character in the array, even if it does not extract any characters from the stream buffer. ios::failbit is set if EOF is encountered before any characters are stored.

AIX Considerations

This function is available for 32-bit applications. The second argument is an int value.

Overload 5

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

Extracts characters from the stream buffer attached to the istream object and stores them in the streambuf, *sb*. The default value of the delim argument is '\n'. Extraction stops when any of the following conditions is true:

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

Overload 6

```
public:istream& get(unsigned char* b, long lim, char delim = '\n')
```

Extracts characters from the stream buffer attached to the istream object and stores them in the byte array beginning at the location pointed to by the first argument and extending for *lim* bytes. The default value of the *delim* argument is '\n'. Extraction stops when either of the following conditions is true:

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

get() always stores a terminating null character in the array, even if it does not extract any characters from the stream buffer. ios::failbit is set if EOF is encountered before any characters are stored.

AIX Considerations

This function is available for 64-bit applications. The second argument is a long value.

Overload 7

```
public:istream& get(char& c)
```

Extracts a single character from the stream buffer attached to the istream object and stores this character in *c*.

Overload 8

```
public:istream& get(signed char& c)
```

Extracts a single character from the stream buffer attached to the istream object and stores this character in *c*.

Overload 9

```
public:istream& get(wchar_t&)
```

Extracts a single wchar_t character from the stream buffer attached to the istream object and stores this character in *c*.

Overload 10

```
public:istream& get(unsigned char* b, int lim, char delim = '\n')
```

Extracts characters from the stream buffer attached to the istream object and stores them in the byte array beginning at the location pointed to by the first argument and extending for *lim* bytes. The default value of the *delim* argument is '\n'. Extraction stops when either of the following conditions is true:

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

get() always stores a terminating null character in the array, even if it does not extract any characters from the stream buffer. ios::failbit is set if EOF is encountered before any characters are stored.

AIX Considerations

This function is available for 32-bit applications. The second argument is an int value.

Overload 11

```
public:istream& get(signed char* b, long lim, char delim = '\n')
```

Extracts characters from the stream buffer attached to the istream object and stores them in the byte array beginning at the location pointed to by the first argument and extending for *lim* bytes. The default value of the *delim* argument is '\n'. Extraction stops when either of the following conditions is true:

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

get() always stores a terminating null character in the array, even if it does not extract any characters from the stream buffer. ios::failbit is set if EOF is encountered before any characters are stored.

AIX Considerations

This function is available for 64-bit applications. The second argument is a long value.

Overload 12

```
public:istream& get(char*, long lim, char delim = '\n')
```

Extracts characters from the stream buffer attached to the istream object and stores them in the byte array beginning at the location pointed to by the first argument and extending for *lim* bytes. The default value of the *delim* argument is '\n'. Extraction stops when either of the following conditions is true:

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

get() always stores a terminating null character in the array, even if it does not extract any characters from the stream buffer. ios::failbit is set if EOF is encountered before any characters are stored.

AIX Considerations

This function is available for 64-bit applications. The second argument is a long value.

get_complicated

Overload 1

```
public:istream& get_complicated(signed char& c)
```

Internal function. Do not use.

Overload 2

```
public:istream& get_complicated(unsigned char& c)
```

Internal function. Do not use.

Overload 3

```
public:istream& get_complicated(char& c)
```

Internal function. Do not use.

getline

Overload 1

```
public:istream&
  getline( unsigned char* b,
           int lim,
           char delim = '\n' )
```

Extracts characters from the stream buffer attached to the istream object and stores them in the byte array beginning at the location pointed to by the first argument and extending for *lim* bytes. The default value of the *delim* argument is '\n'. Extraction stops when either of the following conditions is true:

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

getline() always stores a terminating null character in the array, even if it does not extract any characters from the stream buffer. ios::failbit is set if EOF is encountered before any characters are stored.

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

AIX Considerations

This function is available for 32-bit applications. The second argument is an int value.

Overload 2

```
public:istream&
  getline( unsigned char* b,
           long lim,
           char delim = '\n' )
```

Extracts characters from the stream buffer attached to the istream object and stores them in the byte array beginning at the location pointed to by the first argument and extending for *lim* bytes. The default value of the *delim* argument is '\n'. Extraction stops when either of the following conditions is true:

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

getline() always stores a terminating null character in the array, even if it does not extract any characters from the stream buffer. ios::failbit is set if EOF is encountered before any characters are stored.

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

AIX Considerations

This function is available for 64-bit applications. The second argument is a long value.

Overload 3

```
public:istream& getline(char* b, int lim, char delim = '\n')
```

Extracts characters from the stream buffer attached to the istream object and stores them in the byte array beginning at the location pointed to by the first argument and extending for *lim* bytes. The default value of the *delim* argument is '\n'. Extraction stops when either of the following conditions is true:

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

getline() always stores a terminating null character in the array, even if it does not extract any characters from the stream buffer. ios::failbit is set if EOF is encountered before any characters are stored.

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

AIX Considerations

This function is available for 32-bit applications. The second argument is an int value.

Overload 4

```
public:istream& getline(char* b, long lim, char delim = '\n')
```

Extracts characters from the stream buffer attached to the istream object and stores them in the byte array beginning at the location pointed to by the first argument and extending for *lim* bytes. The default value of the *delim* argument is '\n'. Extraction stops when either of the following conditions is true:

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

getline() always stores a terminating null character in the array, even if it does not extract any characters from the stream buffer. ios::failbit is set if EOF is encountered before any characters are stored.

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

AIX Considerations

This function is available for 64-bit applications. The second argument is a long value.

Overload 5

```
public:istream&
  getline( signed char* b,
           int lim,
           char delim = '\n' )
```

Extracts characters from the stream buffer attached to the istream object and stores them in the byte array beginning at the location pointed to by the first argument and extending for *lim* bytes. The default value of the *delim* argument is '\n'. Extraction stops when either of the following conditions is true:

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

getline() always stores a terminating null character in the array, even if it does not extract any characters from the stream buffer. ios::failbit is set if EOF is encountered before any characters are stored.

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

AIX Considerations

This function is available for 32-bit applications. The second argument is an int value.

Overload 6

```
public:istream&
  getline( signed char* b,
           long lim,
           char delim = '\n' )
```

Extracts characters from the stream buffer attached to the istream object and stores them in the byte array beginning at the location pointed to by the first argument and extending for *lim* bytes. The default value of the *delim* argument is '\n'. Extraction stops when either of the following conditions is true:

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

getline() always stores a terminating null character in the array, even if it does not extract any characters from the stream buffer. ios::failbit is set if EOF is encountered before any characters are stored.

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

AIX Considerations

This function is available for 64-bit applications. The second argument is a long value.

ignore

```
public:istream& ignore(int n = 1, int delim = EOF)
```

Extracts up to *n* characters from the stream buffer attached to the istream object and discards them. ignore() will extract fewer than *n* characters if it encounters *delim* or EOF.

peek

```
public:int peek()
```

peek() calls ipfx(1). If ipfx() returns 0, or if no more input is available from the ultimate producer, peek() returns EOF. Otherwise, it returns the next character in the stream buffer without extracting the character.

read

Overload 1

```
public:istream& read(char* s, long n)
```

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

AIX Considerations

This function is available for 64-bit applications. The second argument is a long value.

Overload 2

```
public:istream& read(signed char* s, int n)
```

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

AIX Considerations

This function is available for 32-bit applications. The second argument is an int value.

Overload 3

```
public:istream& read(unsigned char* s, long n)
```

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

AIX Considerations

This function is available for 64-bit applications. The second argument is a long value.

Overload 4

```
public:istream& read(unsigned char* s, int n)
```

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

AIX Considerations

This function is available for 32-bit applications. The second argument is an int value.

Overload 5

```
public:istream& read(signed char* s, long n)
```

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

AIX Considerations

This function is available for 64-bit applications. The second argument is a long value.

Overload 6

```
public:istream& read(char* s, int n)
```

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

AIX Considerations

This function is available for 32-bit applications. The second argument is an int value.

rs_complicated

Overload 1

```
public:istream& rs_complicated(signed char& c)
```

Internal function. Do not use.

Overload 2

```
public:istream& rs_complicated(char& c)
```

Internal function. Do not use.

Overload 3

```
public:istream& rs_complicated(unsigned char& c)
```

Internal function. Do not use.

eatwhite

```
protected:void eatwhite()
```

Internal function. Do not use.

Input Operators

Input operators supported by istream objects.

operator >>

Overload 1

```
public:istream& operator >>(float&)
```

The input operator converts characters from the stream buffer attached to the input stream according to the C++ lexical conventions.

The following conversions occur for certain string values:

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

Note that if you use thse string values as input in a program compiled with z/OS C/C++, the input operator will not recognize them as floating point numbers and will set ios::badbit in the stream’s error state.

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

Overload 2

```
public:istream& operator >>(char*)
```

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

The input operator always stores a terminating null character in the array, even if an error occurs.

Overload 3

```
public:istream& operator >>(int&)
```

The input operator extracts characters from the stream buffer associated with the input stream and converts them according to the format state of the input stream. The converted characters are then store in the reference location provided. There is no overflow detection on conversion of integral types.

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

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

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

- If these characters are ″0x″ or ″0X″, the subsequent characters are converted to a hexadecimal value.
- If the first character is ″0″ and the second character is not a ″x″ or ″X″, the subsequent characters are converted to an octal value.

- If neither of these cases is true, the characters are converted to a decimal value.

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

Overload 4

```
public:istream& operator >>(long double&)
```

The input operator converts characters from the stream buffer attached to the input stream according to the C++ lexical conventions.

The following conversions occur for certain string values:

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

Note that if you use thse string values as input in a program compiled with z/OS C/C++, the input operator will not recognize them as floating point numbers and will set ios::badbit in the stream's error state.

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

Overload 5

```
public:istream& operator >>(ios & ( * f ) ( ios & ))
```

The following built-in manipulators are accepted by this input operator:

```
ios&   dec(ios&)
ios&   hex(ios&)
ios&   oct(ios &)
```

These manipulators have a specific effect on an istream object beyond extracting their own values. For example, If *ins* is a reference to an istream object, then this statement sets ios::dec:

```
ins >> dec;
```

Overload 6

```
public:istream& operator >>(long long&)
```

The input operator extracts characters from the stream buffer associated with the input stream and converts them according to the format state of the input stream. The converted characters are then store in the reference location provided. There is no overflow detection on conversion of integral types.

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

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

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

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

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

Note: The support for long long is controlled by _LONG_LONG, __EXTENDED__, or the -q(no)longlong option.

Overload 7

```
public:istream& operator >>(long&)
```

The input operator extracts characters from the stream buffer associated with the input stream and converts them according to the format state of the input stream. The converted characters are then store in the reference location provided. There is no overflow detection on conversion of integral types.

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

- ios::oct - the characters are converted to an octal value. Characters are extracted from the stream buffer until a character

that is not an octal digit (a digit from 0 to 7) is encountered. If ios::oct is set and a signed value is encountered, the value is converted into a decimal value.

- ios::dec - the characters are converted to a decimal value. Characters are extracted from the stream buffer until a character that is not a decimal digit (a digit from 0 to 9) is encountered.
- ios::hex - the characters are converted to an hexadecimal value. Characters are extracted from the stream buffer until a character that is not a hexadecimal digit (a digit from 0 to 0 or a letter fromm ″A″ to ″F″, upper or lower case) is encountered. If ios::hex is set and a signed value is encountered, the value is converted into a decimal value.

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

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

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

Overload 8

```
public:istream& operator >>(short&)
```

The input operator extracts characters from the stream buffer associated with the input stream and converts them according to the format state of the input stream. The converted characters are then store in the reference location provided. There is no overflow detection on conversion of integral types.

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

- ios::oct - the characters are converted to an octal value. Characters are extracted from the stream buffer until a character that is not an octal digit (a digit from 0 to 7) is encountered. If ios::oct is set and a signed value is encountered, the value is converted into a decimal value.
- ios::dec - the characters are converted to a decimal value. Characters are extracted from the stream buffer until a character that is not a decimal digit (a digit from 0 to 9) is encountered.
- ios::hex - the characters are converted to an hexadecimal value. Characters are extracted from the stream buffer until a character that is not a hexadecimal digit (a digit from 0 to 0 or a letter

fromm ″A″ to ″F″, upper or lower case) is encountered. If ios::hex is set and a signed value is encountered, the value is converted into a decimal value.

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

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

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

Overload 9

```
public:istream& operator >>(signed char& c)
```

The input operator extracts a character from the stream buffer attached to the input stream and stores it in *c*.

Overload 10

```
public:istream& operator >>(signed char*)
```

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

The input operator always stores a terminating null character in the array, even if an error occurs.

Overload 11

```
public:istream& operator >>(unsigned char*)
```

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

The input operator always stores a terminating null character in the array, even if an error occurs.

Overload 12

```
public:istream& operator >>(streambuf*)
```

For pointers to streambuf objects, the input operator calls ipfx(0). If ipfx(0) returns a nonzero value, the input operator extracts

characters from the stream buffer attached to the istream object and inserts them in the streambuf. Extraction stops when an EOF character is encountered.

The input operator always returns a reference to the istream object.

Overload 13

```
public:istream& operator >>(unsigned int&)
```

The input operator extracts characters from the stream buffer associated with the input stream and converts them according to the format state of the input stream. The converted characters are then store in the reference location provided. There is no overflow detection on conversion of integral types.

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

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

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

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

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

Overload 14

```
public:istream& operator >>(unsigned long long&)
```

The input operator extracts characters from the stream buffer associated with the input stream and converts them according to the format state of the input stream. The converted characters are

then store in the reference location provided. There is no overflow detection on conversion of integral types.

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

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

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

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

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

Note: The support for long long is controlled by _LONG_LONG, __EXTENDED__, or the -q(no)longlong option.

Overload 15

```
public:istream& operator >>(unsigned long&)
```

The input operator extracts characters from the stream buffer associated with the input stream and converts them according to the format state of the input stream. The converted characters are then store in the reference location provided. There is no overflow detection on conversion of integral types.

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

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

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

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

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

Overload 16

```
public:istream& operator >>(unsigned short&)
```

The input operator extracts characters from the stream buffer associated with the input stream and converts them according to the format state of the input stream. The converted characters are then store in the reference location provided. There is no overflow detection on conversion of integral types.

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

- ios::oct - the characters are converted to an octal value. Characters are extracted from the stream buffer until a character that is not an octal digit (a digit from 0 to 7) is encountered. If ios::oct is set and a signed value is encountered, the value is converted into a decimal value.
- ios::dec - the characters are converted to a decimal value. Characters are extracted from the stream buffer until a character that is not a decimal digit (a digit from 0 to 9) is encountered.
- ios::hex - the characters are converted to an hexadecimal value. Characters are extracted from the stream buffer until a character that is not a hexadecimal digit (a digit from 0 to 0 or a letter

fromm ″A″ to ″F″, upper or lower case) is encountered. If ios::hex is set and a signed value is encountered, the value is converted into a decimal value.

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

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

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

Overload 17

```
public:istream& operator >>(wchar_t&)
```

The input operator extracts a wchar_t character from the stream buffer attached to the input stream and stores it in the reference location provided. If ios::skipws is set, the input operator skips leading wchar_t spaces as well as leading char white spaces.

Overload 18

```
public:istream& operator >>(wchar_t*)
```

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

For wchar_t* arrays, if ios::x_width does not equal 0, a maximum of ios::x_width - 1 characters (at 2 bytes each) are extracted. A 2-character space is reserved for the wchar_t terminating null character.

The input operator resets ios::x_width to 0.

The input operator always stores a terminating null character in the array, even if an error occurs. For arrays of wchar_t*, this terminating null character is a wchar_t terminating null character.

Overload 19

```
public:istream& operator >>(unsigned char& c)
```

The input operator extracts a character from the stream buffer attached to the input stream and stores it in *c*.

Overload 20

```
public:istream& operator >>(istream & ( * f ) ( istream & ))
```

The following built-in manipulators are accepted by this input operator:

```
istream&   ws(istream&)
```

These manipulators have a specific effect on an istream object beyond extracting their own values. For example, If *ins* is a reference to an istream object, then this statement extracts white-space characters from the stream buffer attached to *ins*:

```
ins >> ws;
```

Overload 21

```
public:istream& operator >>(double&)
```

The input operator converts characters from the stream buffer attached to the input stream according to the C++ lexical conventions.

The following conversions occur for certain string values:

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

Note that if you use thse string values as input in a program compiled with z/OS C/C++, the input operator will not recognize them as floating point numbers and will set ios::badbit in the stream's error state.

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

Overload 22

```
public:istream& operator >>(char& c)
```

The input operator extracts a character from the stream buffer attached to the input stream and stores it in *c*.

Positioning Functions

Functions that work with the get pointer of the ultimate producer.

putback

```
public:istream& putback(char c)
```

putback() attempts to put an extracted character back into the stream buffer. *c* must equal the character before the get pointer of the stream buffer. Unless some other activity is modifying the stream buffer, this is the last character extracted from the stream buffer. If *c* is not equal to the character before the get pointer, the result of putback() is undefined, and the error state of the input stream may be set. putback() does not call ipfx(), but if the error state of the input stream is nonzero, putback() returns without putting back the character or setting the error state.

seekg

Overload 1

```
public:istream& seekg(streampos p)
```

Sets the get pointer to the position *p*.

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

Overload 2

```
public:istream& seekg(streamoff o, ios::seek_dir d)
```

Sets the get pointer to the position specified by *d* with the offset *o*. The argument *d* can have the following values:

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

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

sync

```
public:int sync()
```

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

tellg

```
public:streampos tellg()
```

Returns the current position of the get pointer of the ultimate producer.

Prefix and Suffix Functions

Functions that are called either before or after extracting characters from the ultimate producer.

ipfx

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

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

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

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

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

Overload 1

```
public:int ipfx(int noskipws = 0)
```

AIX Considerations

This function accepts an int value for 32-bit applications. It is not available for 64-bit applications.

Overload 2

```
public:int ipfx(long noskipws = 0)
```

AIX Considerations

This function accepts a long value for 64-bit applications. It is not available for 32-bit applications.

isfx

```
public:void isfx()
```

Internal function. Do not use.

do_ipfx

Overload 1

```
protected:int do_ipfx(long noskipws)
```

Internal function. Do not use.

AIX Considerations

This function is available for 64-bit applications. It accepts a long argument.

Overload 2

```
protected:int do_ipfx(int noskipws)
```

Internal function. Do not use.

AIX Considerations

This function is available for 32-bit applications. It accepts an int argument.

istream - Inherited Member Functions and Data

Inherited Public Functions

ios	
Definition	**Page Number**
`virtual ~ios()`	143
`int bad() const`	144
`static long bitalloc()`	149
`void clear(int i = 0)`	145
`int eof() const`	145
`int fail() const`	145
`char fill() const`	145
`char fill(char)`	145
`long flags() const`	146
`long flags(long f)`	146
`int good() const`	145

ios	
Definition	**Page Number**
`ios(streambuf*)`	143
`long& iword(int)`	149
`int operator !() const`	149
`operator const void *() const`	149
`operator void *()`	149
`int precision() const`	146
`int precision(int)`	146
`void *& pword(int)`	149
`streambuf* rdbuf()`	149
`int rdstate() const`	145
`long setf(long setbits, long field)`	147
`long setf(long)`	147
`int skip(int i)`	147
`static void sync_with_stdio()`	149
`ostream* tie(ostream* s)`	150
`ostream* tie()`	150
`long unsetf(long)`	147
`int width(int w)`	148
`int width() const`	148
`static long xalloc()`	151
`static int xalloc()`	151

Inherited Public Data

ios	
Definition	**Page Number**
`static const long adjustfield`	143
`static const long basefield`	144
`static const long floatfield`	144

Inherited Protected Functions

ios	
Definition	**Page Number**
`void init(streambuf*)`	149

ios	
Definition	**Page Number**
`ios()`	143
`void setstate(int b)`	145

Inherited Protected Data

ios	
Definition	**Page Number**
`static void ( * stdioflush ) ( )`	151
`int assign_private`	144
`streambuf* bp`	144
`int delbuf`	144
`int isfx_special`	144
`int ispecial`	144
`int osfx_special`	144
`int ospecial`	144
`int state`	144
`char x_fill`	148
`long x_flags`	144
`short x_precision`	148
`ostream* x_tie`	144
`short x_width`	148

istream_withassign

Use this class to assign another stream to an istream object.

Class header file: iostream.h

istream_withassign - Hierarchy List

ios
istream
istream_withassign

istream_withassign - Member Functions and Data by Group

Constructors & Destructor

Objects of the istream_withassign class can be constructed and destructed. They can also be copied.

~istream_withassign

```
public:virtual ~istream_withassign()
```

Destructs an ostream_withassign object.

istream_withassign

```
public:istream_withassign()
```

Creates an istream_withassign object. It does not do any initialization of this object.

operator =

```
public:istream_withassign& operator =(istream_withassign& rhs)
```

The copy constructor.

Assignment Operator

Assignment operators for istream_withassign.

operator =

Overload 1

```
public:istream_withassign& operator =(streambuf*)
```

This assignment operator takes a pointer to a streambuf object as its argument. It associates this streambuf object with the istream_withassign object that is on the left side of the assignment operator.

Overload 2

```
public:istream_withassign& operator =(istream&)
```

This assignment operator takes an istream objects as its argument. It associates the stream buffer attached to the input stream with the istream_withassign object that is on the left side of the assignment operator.

istream_withassign - Inherited Member Functions and Data

Inherited Public Functions

ios	
Definition	**Page Number**
`virtual ~ios()`	143
`int bad() const`	144
`static long bitalloc()`	149
`void clear(int i = 0)`	145
`int eof() const`	145
`int fail() const`	145
`char fill() const`	145
`char fill(char)`	145
`long flags(long f)`	146
`long flags() const`	146
`int good() const`	145
`ios(streambuf*)`	143

ios	
Definition	**Page Number**
`long& iword(int)`	149
`int operator !() const`	149
`operator const void *() const`	149
`operator void *()`	149
`int precision() const`	146
`int precision(int)`	146
`void *& pword(int)`	149
`streambuf* rdbuf()`	149
`int rdstate() const`	145
`long setf(long)`	147
`long setf(long setbits, long field)`	147
`int skip(int i)`	147
`static void sync_with_stdio()`	149
`ostream* tie(ostream* s)`	150
`ostream* tie()`	150
`long unsetf(long)`	147
`int width(int w)`	148
`int width() const`	148
`static long xalloc()`	151
`static int xalloc()`	151

istream	
Definition	**Page Number**
`virtual ~istream()`	165
`long gcount()`	165
`int gcount()`	165
`istream& get(unsigned char& c)`	166
`istream& get(char*, int lim, char delim = '\n')`	166
`istream& get(signed char* b, long lim, char delim = '\n')`	166
`istream& get(char& c)`	166
`istream& get(streambuf& sb, char delim = '\n')`	166
`istream& get(wchar_t&)`	166
`istream& get(signed char* b, int lim, char delim = '\n')`	166

istream	
Definition	**Page Number**
istream& get(signed char& c)	166
int get()	166
istream& get(char*, long lim, char delim = '\n')	166
istream& get(unsigned char* b, long lim, char delim = '\n')	166
istream& get(unsigned char* b, int lim, char delim = '\n')	166
istream& get_complicated(unsigned char& c)	169
istream& get_complicated(char& c)	169
istream& get_complicated(signed char& c)	169
istream& getline(char* b, int lim, char delim = '\n')	169
istream& getline(unsigned char* b, long lim, char delim = '\n')	169
istream& getline(unsigned char* b, int lim, char delim = '\n')	169
istream& getline(char* b, long lim, char delim = '\n')	169
istream& getline(signed char* b, int lim, char delim = '\n')	169
istream& getline(signed char* b, long lim, char delim = '\n')	169
istream& ignore(int n = 1, int delim = EOF)	172
int ipfx(int noskipws = 0)	185
int ipfx(long noskipws = 0)	185
void isfx()	186
istream(int size, char*, int sk = 1)	165
istream(int fd, int sk = 1, ostream* t = 0)	165
istream(streambuf*, int sk, ostream* t = 0)	165
istream(streambuf*)	165
istream& operator >>(unsigned char& c)	174
istream& operator >>(wchar_t*)	174
istream& operator >>(unsigned short&)	174
istream& operator >>(istream & (* f) (istream &))	174

istream	
Definition	**Page Number**
istream& operator >>(unsigned long long&)	174
istream& operator >>(char*)	174
istream& operator >>(unsigned int&)	174
istream& operator >>(long double&)	174
istream& operator >>(int&)	174
istream& operator >>(long long&)	174
istream& operator >>(ios & (* f) (ios &))	174
istream& operator >>(double&)	174
istream& operator >>(signed char& c)	174
istream& operator >>(streambuf*)	174
istream& operator >>(char& c)	174
istream& operator >>(signed char*)	174
istream& operator >>(unsigned char*)	174
istream& operator >>(unsigned long&)	174
istream& operator >>(wchar_t&)	174
istream& operator >>(float&)	174
istream& operator >>(short&)	174
istream& operator >>(long&)	174
int peek()	172
istream& putback(char c)	184
istream& read(signed char* s, long n)	172
istream& read(char* s, int n)	172
istream& read(unsigned char* s, int n)	172
istream& read(signed char* s, int n)	172
istream& read(unsigned char* s, long n)	172
istream& read(char* s, long n)	172
istream& rs_complicated(unsigned char& c)	174
istream& rs_complicated(char& c)	174
istream& rs_complicated(signed char& c)	174
istream& seekg(streamoff o, ios::seek_dir d)	184
istream& seekg(streampos p)	184
int sync()	185
streampos tellg()	185

Inherited Public Data

ios	
Definition	**Page Number**
static const long adjustfield	143
static const long basefield	144
static const long floatfield	144

Inherited Protected Functions

istream	
Definition	**Page Number**
int do_ipfx(int noskipws)	186
int do_ipfx(long noskipws)	186
void eatwhite()	174
istream()	165

ios	
Definition	**Page Number**
void init(streambuf*)	149
ios()	143
void setstate(int b)	145

Inherited Protected Data

ios	
Definition	**Page Number**
static void (* stdioflush) ( )	151
int assign_private	144
streambuf* bp	144
int delbuf	144
int isfx_special	144
int ispecial	144
int osfx_special	144
int ospecial	144
int state	144
char x_fill	148
long x_flags	144

ios	
Definition	**Page Number**
`short x_precision`	148
`ostream* x_tie`	144
`short x_width`	148

istrstream

istrstream is the class that specializes istream to use a strstreambuf for extraction from arrays of characters in memory. You can create an istrstream object by associating the object with a previously allocated array of characters. You can then read input from it and apply other operations to it just as you would to another type of stream.

Class header file: strstream.h

istrstream - Hierarchy List

ios

strstreambase

istrstream

istrstream - Member Functions and Data by Group

Constructors & Destructor

Objects of the istrstream class can be constructed and destructed.

~istrstream

```
public:~istrstream()
```

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

istrstream

Overload 1

```
public:istrstream(const char* str)
```

This constructor specifies that characters should be extracted from the null-terminated string that is pointed to by str. You can use the istream::seekg() function to reposition the get pointer in this string.

Overload 2

```
public:istrstream(const signed char* str)
```

This constructor specifies that characters should be extracted from the null-terminated string that is pointed to by str. You can use the istream::seekg() function to reposition the get pointer in this string.

Overload 3

```
public:istrstream(char* str, long size)
```

This constructor specifies that characters should be extracted from the array of bytes that starts at the position pointed to by str and

has a length of size bytes. You can use the istream::seekg() function to reposition the get pointer anywhere in this array.

AIX Considerations

This function is available for 64-bit applications. The second argument is a long value.

Overload 4

```
public:istrstream(signed char* str, long size)
```

This constructor specifies that characters should be extracted from the array of bytes that starts at the position pointed to by str and has a length of size bytes. You can use the istream::seekg() function to reposition the get pointer anywhere in this array.

AIX Considerations

This function is available for 64-bit applications. The second argument is a long value.

Overload 5

```
public:istrstream(const signed char* str, int size)
```

This constructor specifies that characters should be extracted from the array of bytes that starts at the position pointed to by str and has a length of size bytes. You can use the istream::seekg() function to reposition the get pointer anywhere in this array.

AIX Considerations

This function is available for 32-bit applications. The second argument is an int value.

Overload 6

```
public:istrstream(const signed char* str, long size)
```

This constructor specifies that characters should be extracted from the array of bytes that starts at the position pointed to by str and has a length of size bytes. You can use the istream::seekg() function to reposition the get pointer anywhere in this array.

AIX Considerations

This function is available for 64-bit applications. The second argument is a long value.

Overload 7

```
public:istrstream(const unsigned char* str)
```

This constructor specifies that characters should be extracted from the null-terminated string that is pointed to by str. You can use the istream::seekg() function to reposition the get pointer in this string.

Overload 8

```
public:istrstream(const unsigned char* str, long size)
```

This constructor specifies that characters should be extracted from the array of bytes that starts at the position pointed to by str and has a length of size bytes. You can use the istream::seekg() function to reposition the get pointer anywhere in this array.

AIX Considerations

This function is available for 64-bit applications. The second argument is a long value.

Overload 9

```
public:istrstream(const unsigned char* str, int size)
```

This constructor specifies that characters should be extracted from the array of bytes that starts at the position pointed to by str and has a length of size bytes. You can use the istream::seekg() function to reposition the get pointer anywhere in this array.

AIX Considerations

This function is available for 32-bit applications. The second argument is an int value.

Overload 10

```
public:istrstream(const char* str, int size)
```

This constructor specifies that characters should be extracted from the array of bytes that starts at the position pointed to by str and has a length of size bytes. You can use the istream::seekg() function to reposition the get pointer anywhere in this array.

AIX Considerations

This function is available for 32-bit applications. The second argument is an int value.

Overload 11

```
public:istrstream(signed char* str)
```

This constructor specifies that characters should be extracted from the null-terminated string that is pointed to by str. You can use the istream::seekg() function to reposition the get pointer in this string.

Overload 12

```
public:istrstream(unsigned char* str)
```

This constructor specifies that characters should be extracted from the null-terminated string that is pointed to by str. You can use the istream::seekg() function to reposition the get pointer in this string.

Overload 13

```
public:istrstream(unsigned char* str, int size)
```

This constructor specifies that characters should be extracted from the array of bytes that starts at the position pointed to by str and has a length of size bytes. You can use the istream::seekg() function to reposition the get pointer anywhere in this array.

AIX Considerations

This function is available for 32-bit applications. The second argument is an int value.

Overload 14

```
public:istrstream(unsigned char* str, long size)
```

This constructor specifies that characters should be extracted from the array of bytes that starts at the position pointed to by str and

has a length of size bytes. You can use the istream::seekg() function to reposition the get pointer anywhere in this array.

AIX Considerations

This function is available for 64-bit applications. The second argument is a long value.

Overload 15

```
public:istrstream(signed char* str, int size)
```

This constructor specifies that characters should be extracted from the array of bytes that starts at the position pointed to by str and has a length of size bytes. You can use the istream::seekg() function to reposition the get pointer anywhere in this array.

AIX Considerations

This function is available for 32-bit applications. The second argument is an int value.

Overload 16

```
public:istrstream(const char* str, long size)
```

This constructor specifies that characters should be extracted from the array of bytes that starts at the position pointed to by str and has a length of size bytes. You can use the istream::seekg() function to reposition the get pointer anywhere in this array.

AIX Considerations

This function is available for 64-bit applications. The second argument is a long value.

Overload 17

```
public:istrstream(char* str, int size)
```

This constructor specifies that characters should be extracted from the array of bytes that starts at the position pointed to by str and has a length of size bytes. You can use the istream::seekg() function to reposition the get pointer anywhere in this array.

AIX Considerations

This function is available for 32-bit applications. The second argument is an int value.

Overload 18

```
public:istrstream(char* str)
```

This constructor specifies that characters should be extracted from the null-terminated string that is pointed to by str. You can use the istream::seekg() function to reposition the get pointer in this string.

istrstream - Inherited Member Functions and Data

Inherited Public Functions

ios	
Definition	**Page Number**
`virtual ~ios()`	143

ios	
Definition	**Page Number**
int bad() const	144
static long bitalloc()	149
void clear(int i = 0)	145
int eof() const	145
int fail() const	145
char fill() const	145
char fill(char)	145
long flags() const	146
long flags(long f)	146
int good() const	145
ios(streambuf*)	143
long& iword(int)	149
int operator !() const	149
operator const void *() const	149
operator void *()	149
int precision() const	146
int precision(int)	146
void *& pword(int)	149
streambuf* rdbuf()	149
int rdstate() const	145
long setf(long setbits, long field)	147
long setf(long)	147
int skip(int i)	147
static void sync_with_stdio()	149
ostream* tie(ostream* s)	150
ostream* tie()	150
long unsetf(long)	147
int width(int w)	148
int width() const	148
static int xalloc()	151
static long xalloc()	151

strstreambase	
Definition	**Page Number**
strstreambuf* rdbuf()	253

Inherited Public Data

ios	
Definition	**Page Number**
static const long adjustfield	143
static const long basefield	144
static const long floatfield	144

Inherited Protected Functions

strstreambase	
Definition	**Page Number**
~strstreambase()	253
strstreambase(char*, long, char*)	253
strstreambase(char*, int, char*)	253
strstreambase()	253

ios	
Definition	**Page Number**
void init(streambuf*)	149
ios()	143
void setstate(int b)	145

Inherited Protected Data

ios	
Definition	**Page Number**
static void (* stdioflush) ( )	151
int assign_private	144
streambuf* bp	144
int delbuf	144
int isfx_special	144
int ispecial	144
int osfx_special	144

ios	
Definition	**Page Number**
`int ospecial`	144
`int state`	144
`char x_fill`	148
`long x_flags`	144
`short x_precision`	148
`ostream* x_tie`	144
`short x_width`	148

ofstream

This class specializes the ostream class for use with files.

Class header file: fstream.h

ofstream - Hierarchy List

ios

fstreambase

ofstream

ofstream - Member Functions and Data by Group

Constructors & Destructor

Objects of the ofstream class can be constructed and destructed.

~ofstream

```
public:~ofstream()
```

Destructs an ofstream object.

ofstream

Constructs an object of this class.

Overload 1

```
public:ofstream(int fd, char* p, int l)
```

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

AIX Considerations

This function is available for 32-bit applications. The third argument is an int value.

Overload 2

```
public:ofstream(int fd)
```

Constructs an ofstream object that is attached to the file descriptor *fd*. If *fd* is not open, ios::failbit is set in the format state of the ofstream object.

Overload 3

```
public:ofstream(int fd, char* p, long l)
```

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

AIX Considerations

This function is available for 64-bit applications. The third argument is a long value.

Overload 5

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

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

Overload 6

```
public:ofstream()
```

Constructs an unopened ofstream object.

Filebuf Functions

rdbuf

```
public:filebuf* rdbuf()
```

Returns a pointer to the filebuf object that is attached to the ofstream object.

Open Functions

Opens the file.

open

Opens the specified file.

Overload 1

```
public:void
  open( const char* name,
        int mode = ios::out,
        int prot = filebuf::openprot )
```

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

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

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

ofstream - Inherited Member Functions and Data

Inherited Public Functions

fstreambase	
Definition	**Page Number**
`~fstreambase()`	133
`void attach(FILE* fp)`	134
`void attach(int fd)`	134
`void close()`	134
`int detach()`	134
`fstreambase(int fd, char* p, long l)`	133
`fstreambase( const char* name, const char* attr, int mode, int prot = filebuf::openprot )`	133
`fstreambase()`	133
`fstreambase(int fd, char* p, int l)`	133
`fstreambase( const char* name, int mode, int prot = filebuf::openprot )`	133
`fstreambase(int fd)`	133
`void open( const char* name, int mode, int prot = filebuf::openprot )`	135
`void open( const char* name, const char* attr, int mode, int prot = filebuf::openprot )`	135
`void setbuf(char* p, int l)`	135
`void setbuf(char* p, long l)`	135

ios	
Definition	**Page Number**
`virtual ~ios()`	143

ios	
Definition	**Page Number**
int bad() const	144
static long bitalloc()	149
void clear(int i = 0)	145
int eof() const	145
int fail() const	145
char fill() const	145
char fill(char)	145
long flags(long f)	146
long flags() const	146
int good() const	145
ios(streambuf*)	143
long& iword(int)	149
int operator !() const	149
operator const void *() const	149
operator void *()	149
int precision() const	146
int precision(int)	146
void *& pword(int)	149
streambuf* rdbuf()	149
int rdstate() const	145
long setf(long setbits, long field)	147
long setf(long)	147
int skip(int i)	147
static void sync_with_stdio()	149
ostream* tie()	150
ostream* tie(ostream* s)	150
long unsetf(long)	147
int width() const	148
int width(int w)	148
static int xalloc()	151
static long xalloc()	151

Inherited Public Data

ios	
Definition	**Page Number**
`static const long adjustfield`	143
`static const long basefield`	144
`static const long floatfield`	144

Inherited Protected Functions

ios	
Definition	**Page Number**
`void init(streambuf*)`	149
`ios()`	143
`void setstate(int b)`	145

fstreambase	
Definition	**Page Number**
`void verify(int)`	135

Inherited Protected Data

ios	
Definition	**Page Number**
`static void ( * stdioflush ) ( )`	151
`int assign_private`	144
`streambuf* bp`	144
`int delbuf`	144
`int isfx_special`	144
`int ispecial`	144
`int osfx_special`	144
`int ospecial`	144
`int state`	144
`char x_fill`	148
`long x_flags`	144
`short x_precision`	148
`ostream* x_tie`	144
`short x_width`	148

ostream

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

```
outs << x;
```

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

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

The output operator is defined for the following types:

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

You can also define output operators for your own types.

Class header file: iostream.h

ostream - Hierarchy List

ios

ostream

ostream_withassign

ostream - Member Functions and Data by Group

Constructors & Destructor

Objects of the ostream class can be constructed and destructed.

~ostream

```
public:virtual ~ostream()
```

Destructs an ostream object.

ostream

Overload 1

```
public:ostream(streambuf*)
```

This constructor takes a single argument which is a pointer to a streambuf object. This constructor creates an ostream object that is attached to the streambuf object pointed to by the argument. The format variables are initialized to their defaults.

Overload 2

```
public:ostream(int fd)
```

This constructor is obsolete; do not use it.

Overload 3

```
public:ostream(int size, char*)
```

This constructor is obsolete; do not use it.

Overload 4

```
protected:ostream()
```

This constructor is obsolete; do not use it.

Insertion Functions

You can use the insertion functions to insert characters into a stream buffer as a sequence of bytes.

complicated_put

```
public:ostream& complicated_put(char c)
```

flush

```
public:ostream& flush()
```

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

ls_complicated

Overload 1

```
public:ostream& ls_complicated(char)
```

Internal function. Do not use.

Overload 2

```
public:ostream& ls_complicated(signed char)
```

Internal function. Do not use.

Overload 3

```
public:ostream& ls_complicated(unsigned char)
```

Internal function. Do not use.

put

```
public:ostream& put(char c)
```

Inserts *c* into the stream buffer attached to the output stream. put() sets the error state of the output stream if the insertion fails.

write

Overload 1

```
public:ostream& write(const signed char* s, int n)
```

Inserts *n* characters that begin at the position pointed to by *s*. This array of characters does not need to end with a null character.

Overload 2

```
public:ostream& write(const char* s, int n)
```

Inserts *n* characters that begin at the position pointed to by *s*. This array of characters does not need to end with a null character.

Overload 3

```
public:ostream& write(const unsigned char* s, int n)
```

Inserts *n* characters that begin at the position pointed to by *s*. This array of characters does not need to end with a null character.

Output operators

The output operator calls the output prefix function opfx() before inserting characters into a stream buffer, and calls the output suffix function osfx() after inserting characters.

operator <<

Overload 1

```
public:ostream& operator <<(const unsigned char*)
```

The output operator inserts all the characters in the string into the stream buffer with the exception of the null character that terminates the string.

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

Overload 2

```
public:ostream& operator <<(const char*)
```

The output operator inserts all the characters in the string into the stream buffer with the exception of the null character that terminates the string.

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

Overload 3

```
public:ostream& operator <<(const void*)
```

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

Overload 4

```
public:ostream& operator <<(ios & ( * f ) ( ios & ))
```

The following built-in manipulators are accepted by this output operator:

```
ios&   dec(ios&)
ios&   hex(ios&)
ios&   oct(ios&)
```

These manipulators have a specific effect on an ostream object beyond inserting their own values. For example, If *outs* is a reference to an ostream object, then this statement sets ios::dec:

```
outs << dec;
```

Overload 5

```
public:ostream& operator <<(unsigned char c)
```

The output operator inserts the character into the stream buffer without performing any conversion on it.

Overload 6

```
public:ostream& operator <<(unsigned long)
```

The output operator converts the integral value according to the format state of the output stream and inserts characters into the stream buffer associated with the output stream. There is no overflow detection on conversion of integral types.

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

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

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

- If the integral type is negative, a negative sign ″-″ is inserted before the decimal digits
- If the integral type is equal to 0, the single digit 0 is inserted
- If the integral type is positive and ios::showpos is set, a positive sign ″+″ is inserted before the decimal digits.

Overload 7

```
public:ostream& operator <<(long long)
```

The output operator converts the integral value according to the format state of the output stream and inserts characters into the stream buffer associated with the output stream. There is no overflow detection on conversion of integral types.

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

- If ios::oct is set, the integral type is converted to a series of octal digits. If ios::showbase is set, ″0″ is inserted into the stream buffer before the octal digits. If the value being inserted is equal to 0, a single ″0″ is inserted, not ″00″.
- If ios::dec is set, the integral type is converted to a series of decimal digits.

- If ios::hex is set, the integral type is converted to a series of hexadecimal digits. If ios::showbase is set, ″0x″ (or ″0X″ if ios::uppercase is set) is inserted into the stream buffer before the hexadecimal digits.

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

- If the integral type is negative, a negative sign ″-″ is inserted before the decimal digits
- If the integral type is equal to 0, the single digit 0 is inserted
- If the integral type is positive and ios::showpos is set, a positive sign ″+″ is inserted before the decimal digits.

Note: The support for long long is controlled by _LONG_LONG, __EXTENDED__, or the -q(no)longlong option.

Overload 8

```
public:ostream& operator <<(unsigned int a)
```

The output operator converts the integral value according to the format state of the output stream and inserts characters into the stream buffer associated with the output stream. There is no overflow detection on conversion of integral types.

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

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

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

- If the integral type is negative, a negative sign ″-″ is inserted before the decimal digits
- If the integral type is equal to 0, the single digit 0 is inserted
- If the integral type is positive and ios::showpos is set, a positive sign ″+″ is inserted before the decimal digits.

Overload 9

```
public:ostream& operator <<(double)
```

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

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

- width() - if this returns 0, the argument is inserted without any fill characters. If the return value is greater than the number of characters needed to represent the argument, extra fill characters are inserted so that the total number of characters inserted is equal to the return value.

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

- If ios::scientific is set, the argument is converted to scientific notation with one digit before the decimal, and the number of digits after the decimal equal to the value returned by precision(). The exponent begins with a lowercase ″e″ unless ios::uppercase is set, in which case the exponent begins with an uppercase ″E″.
- If ios::fixed is set, the argument is converted to fixed notation, with the number of digits after the decimal point equal to the value returned by precision().
- If neither ios::fixed nor ios::scientific is set, the conversion depends upon the value of the argument. If ios::uppercase is set, the exponents of values in scientific notation begin with an uppercase ″E″.

Overload 10

```
public:ostream& operator <<(short i)
```

The output operator converts the integral value according to the format state of the output stream and inserts characters into the stream buffer associated with the output stream. There is no overflow detection on conversion of integral types.

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

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

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

- If the integral type is negative, a negative sign ″-″ is inserted before the decimal digits
- If the integral type is equal to 0, the single digit 0 is inserted
- If the integral type is positive and ios::showpos is set, a positive sign ″+″ is inserted before the decimal digits.

Overload 11

```
public:ostream& operator <<(long double)
```

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

- precision() - returns the number of significant digits that appear after the decimal. The default value is 6.
- width() - if this returns 0, the argument is inserted without any fill characters. If the return value is greater than the number of characters needed to represent the argument, extra fill characters are inserted so that the total number of characters inserted is equal to the return value.

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

- If ios::scientific is set, the argument is converted to scientific notation with one digit before the decimal, and the number of digits after the decimal equal to the value returned by precision(). The exponent begins with a lowercase ″e″ unless ios::uppercase is set, in which case the exponent begins with an uppercase ″E″.
- If ios::fixed is set, the argument is converted to fixed notation, with the number of digits after the decimal point equal to the value returned by precision().
- If neither ios::fixed nor ios::scientific is set, the conversion depends upon the value of the argument. If ios::uppercase is set, the exponents of values in scientific notation begin with an uppercase ″E″.

Overload 12

```
public:ostream& operator <<(int a)
```

The output operator converts the integral value according to the format state of the output stream and inserts characters into the stream buffer associated with the output stream. There is no overflow detection on conversion of integral types.

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

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

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

- If the integral type is negative, a negative sign ″-″ is inserted before the decimal digits
- If the integral type is equal to 0, the single digit 0 is inserted

- If the integral type is positive and ios::showpos is set, a positive sign ″+″ is inserted before the decimal digits.

Overload 13

```
public:ostream& operator <<(long)
```

The output operator converts the integral value according to the format state of the output stream and inserts characters into the stream buffer associated with the output stream. There is no overflow detection on conversion of integral types.

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

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

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

- If the integral type is negative, a negative sign ″-″ is inserted before the decimal digits
- If the integral type is equal to 0, the single digit 0 is inserted
- If the integral type is positive and ios::showpos is set, a positive sign ″+″ is inserted before the decimal digits.

Overload 14

```
public:ostream& operator <<(unsigned long long)
```

The output operator converts the integral value according to the format state of the output stream and inserts characters into the stream buffer associated with the output stream. There is no overflow detection on conversion of integral types.

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

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

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

- If the integral type is negative, a negative sign ″-″ is inserted before the decimal digits
- If the integral type is equal to 0, the single digit 0 is inserted
- If the integral type is positive and ios::showpos is set, a positive sign ″+″ is inserted before the decimal digits.

Note: The support for long long is controlled by _LONG_LONG, __EXTENDED__, or the -q(no)longlong option.

Overload 15

```
public:ostream& operator <<(unsigned short i)
```

The output operator converts the integral value according to the format state of the output stream and inserts characters into the stream buffer associated with the output stream. There is no overflow detection on conversion of integral types.

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

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

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

- If the integral type is negative, a negative sign ″-″ is inserted before the decimal digits
- If the integral type is equal to 0, the single digit 0 is inserted
- If the integral type is positive and ios::showpos is set, a positive sign ″+″ is inserted before the decimal digits.

Overload 16

```
public:ostream& operator <<(const wchar_t*)
```

The output operator converts the wchar_t string to its equivalent multibyte character string, and then inserts it into the stream buffer with the exception of the null character that terminates the string.

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

Overload 17

```
public:ostream& operator <<(signed char c)
```

The output operator inserts the character into the stream buffer without performing any conversion on it.

Overload 18

```
public:ostream& operator <<(float)
```

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

- precision() - returns the number of significant digits that appear after the decimal. The default value is 6.
- width() - if this returns 0, the argument is inserted without any fill characters. If the return value is greater than the number of characters needed to represent the argument, extra fill characters are inserted so that the total number of characters inserted is equal to the return value.

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

- If ios::scientific is set, the argument is converted to scientific notation with one digit before the decimal, and the number of digits after the decimal equal to the value returned by precision(). The exponent begins with a lowercase ″e″ unless ios::uppercase is set, in which case the exponent begins with an uppercase ″E″.
- If ios::fixed is set, the argument is converted to fixed notation, with the number of digits after the decimal point equal to the value returned by precision().
- If neither ios::fixed nor ios::scientific is set, the conversion depends upon the value of the argument. If ios::uppercase is set, the exponents of values in scientific notation begin with an uppercase ″E″.

Overload 19

```
public:ostream& operator <<(ostream & ( * f ) ( ostream & ))
```

The following built-in manipulators are accepted by this output operator:

```
ostream&   endl(ostream&)
ostream&   ends(ostream&)
ostream&   flush(ostream&)
```

These manipulators have a specific effect on an ostream object beyond inserting their own values. For example, If *outs* is a reference to an ostream object, then this statement inserts a newline character and calls flush():

```
outs << endl;
```

This statement inserts a null character:

```
outs << ends;
```

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

```
outs << flush;
```

Overload 20

```
public:ostream& operator <<(wchar_t)
```

The output operator inserts the character into the stream buffer without performing any conversion on it.

Overload 21

```
public:ostream& operator <<(streambuf*)
```

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

Overload 22

```
public:ostream& operator <<(const signed char*)
```

The output operator inserts all the characters in the string into the stream buffer with the exception of the null character that terminates the string.

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

Overload 23

```
public:ostream& operator <<(char c)
```

The output operator inserts the character into the stream buffer without performing any conversion on it.

Positioning Functions

seekp

Functions that work with the put pointer of the ultimate consumer.

Overload 1

```
public:ostream& seekp(streampos p)
```

Repositions the put pointer of the ultimate consumer. Sets the put pointer to the position *p*.

Overload 2

```
public:ostream& seekp(streamoff o, ios::seek_dir d)
```

Repositions the put pointer of the ultimate consumer. Sets the put pointer to the position specified by *d* with the offset of *o*. The seek dir, *d*, can have the following values:

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

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

tellp

```
public:streampos tellp()
```

Returns the current position of the put pointer of the stream buffer that is attached to the output stream.

Prefix and Suffix Functions

Functions that are called either before or after inserting characters into the ultimate consumer.

opfx

```
public:int opfx()
```

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

osfx

```
public:void osfx()
```

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

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

do_opfx

```
protected:int do_opfx()
```

Internal function. Do not use.

do_osfx

```
protected:void do_osfx()
```

Internal function. Do not use.

ostream - Inherited Member Functions and Data

Inherited Public Functions

ios	
Definition	**Page Number**
`virtual ~ios()`	143
`int bad() const`	144
`static long bitalloc()`	149
`void clear(int i = 0)`	145
`int eof() const`	145
`int fail() const`	145
`char fill() const`	145

ios	
Definition	**Page Number**
char fill(char)	145
long flags() const	146
long flags(long f)	146
int good() const	145
ios(streambuf*)	143
long& iword(int)	149
int operator !() const	149
operator const void *() const	149
operator void *()	149
int precision(int)	146
int precision() const	146
void *& pword(int)	149
streambuf* rdbuf()	149
int rdstate() const	145
long setf(long setbits, long field)	147
long setf(long)	147
int skip(int i)	147
static void sync_with_stdio()	149
ostream* tie()	150
ostream* tie(ostream* s)	150
long unsetf(long)	147
int width(int w)	148
int width() const	148
static int xalloc()	151
static long xalloc()	151

Inherited Public Data

ios	
Definition	**Page Number**
static const long adjustfield	143
static const long basefield	144
static const long floatfield	144

Inherited Protected Functions

ios	
Definition	**Page Number**
`void init(streambuf*)`	149
`ios()`	143
`void setstate(int b)`	145

Inherited Protected Data

ios	
Definition	**Page Number**
`static void ( * stdioflush ) ( )`	151
`int assign_private`	144
`streambuf* bp`	144
`int delbuf`	144
`int isfx_special`	144
`int ispecial`	144
`int osfx_special`	144
`int ospecial`	144
`int state`	144
`char x_fill`	148
`long x_flags`	144
`short x_precision`	148
`ostream* x_tie`	144
`short x_width`	148

ostream_withassign

Use this class to assign another stream to an ostream object.

Class header file: iostream.h

ostream_withassign - Hierarchy List

ios
ostream
ostream_withassign

ostream_withassign - Member Functions and Data by Group

Constructors & Destructor

Objects of the ostream_withassign class can be constructed and destructed. They can also be copied.

~ostream_withassign

```
public:virtual ~ostream_withassign()
```

Destructs an ostream_withassign object.

operator =

```
public:ostream_withassign& operator =(ostream_withassign& rhs)
```

Copy constructor.

ostream_withassign

```
public:ostream_withassign()
```

Constructs an ostream_withassign object. It does not do any initialization on the object.

Assignment Operator

Assignment operators for ostream_withassign.

operator =

Overload 1

```
public:ostream_withassign& operator =(streambuf*)
```

This assignment operator takes a pointer to a streambuf object as its argument. It associates the streambuf with the ostream_withassign object that is on the left side of the assignment operator.

Overload 2

```
public:ostream_withassign& operator =(ostream&)
```

This assignment operator takes a reference to an ostream object as its argument. It associates the streambuf attached to the output stream with the ostream_withassign object that is on the left side of the assignment operator.

ostream_withassign - Inherited Member Functions and Data

Inherited Public Functions

ios	
Definition	**Page Number**
`virtual ~ios()`	143
`int bad() const`	144
`static long bitalloc()`	149
`void clear(int i = 0)`	145
`int eof() const`	145
`int fail() const`	145

ios	
Definition	**Page Number**
char fill() const	145
char fill(char)	145
long flags(long f)	146
long flags() const	146
int good() const	145
ios(streambuf*)	143
long& iword(int)	149
int operator !() const	149
operator const void *() const	149
operator void *()	149
int precision(int)	146
int precision() const	146
void *& pword(int)	149
streambuf* rdbuf()	149
int rdstate() const	145
long setf(long setbits, long field)	147
long setf(long)	147
int skip(int i)	147
static void sync_with_stdio()	149
ostream* tie()	150
ostream* tie(ostream* s)	150
long unsetf(long)	147
int width(int w)	148
int width() const	148
static long xalloc()	151
static int xalloc()	151

ostream	
Definition	**Page Number**
virtual ~ostream()	205
ostream& complicated_put(char c)	206
ostream& flush()	206
ostream& ls_complicated(unsigned char)	206

ostream	
Definition	**Page Number**
ostream& ls_complicated(signed char)	206
ostream& ls_complicated(char)	206
ostream& operator <<(double)	207
ostream& operator <<(long)	207
ostream& operator <<(signed char c)	207
ostream& operator <<(unsigned short i)	207
ostream& operator <<(streambuf*)	207
ostream& operator <<(const signed char*)	207
ostream& operator <<(char c)	207
ostream& operator <<(short i)	207
ostream& operator <<(const unsigned char*)	207
ostream& operator <<(long double)	207
ostream& operator <<(float)	207
ostream& operator <<(const void*)	207
ostream& operator <<(unsigned long long)	207
ostream& operator <<(unsigned int a)	207
ostream& operator <<(unsigned char c)	207
ostream& operator <<(unsigned long)	207
ostream& operator <<(long long)	207
ostream& operator <<(ios & (* f) (ios &))	207
ostream& operator <<(const char*)	207
ostream& operator <<(int a)	207
ostream& operator <<(ostream & (* f) (ostream &))	207
ostream& operator <<(wchar_t)	207
ostream& operator <<(const wchar_t*)	207
int opfx()	216
void osfx()	216
ostream(streambuf*)	205
ostream(int fd)	205
ostream(int size, char*)	205
ostream& put(char c)	206
ostream& seekp(streamoff o, ios::seek_dir d)	215
ostream& seekp(streampos p)	215

ostream	
Definition	**Page Number**
streampos tellp()	216
ostream& write(const char* s, int n)	206
ostream& write(const unsigned char* s, int n)	206
ostream& write(const signed char* s, int n)	206

Inherited Public Data

ios	
Definition	**Page Number**
static const long adjustfield	143
static const long basefield	144
static const long floatfield	144

Inherited Protected Functions

ostream	
Definition	**Page Number**
int do_opfx()	216
void do_osfx()	216
ostream()	205

ios	
Definition	**Page Number**
void init(streambuf*)	149
ios()	143
void setstate(int b)	145

Inherited Protected Data

ios	
Definition	**Page Number**
static void (* stdioflush) ( )	151
int assign_private	144
streambuf* bp	144
int delbuf	144
int isfx_special	144

ios	
Definition	**Page Number**
`int ispecial`	144
`int osfx_special`	144
`int ospecial`	144
`int state`	144
`char x_fill`	148
`long x_flags`	144
`short x_precision`	148
`ostream* x_tie`	144
`short x_width`	148

ostrstream

ostrstream is the class that specializes ostream to use a strstreambuf for insertion into arrays of characters in memory. You can create an ostrstream object by associating the object with a previously allocated array of characters. You can then write to it and apply other operations to it just as you would to another type of stream.

Class header file: strstream.h

ostrstream - Hierarchy List

ios

strstreambase

ostrstream

ostrstream - Member Functions and Data by Group

Constructors & Destructor

Objects of the ostrstream class can be constructed and destructed.

~ostrstream

```
public:~ostrstream()
```

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

ostrstream

Overload 1

```
public:ostrstream(signed char* str, int size, int = ios::out)
```

This constructor specifies that the stream buffer that is attached to the ostrstream object consists of an array that starts at the position pointed to by str with a length of size bytes. If ios::ate or ios::app is set, str points to a null-terminated string and insertions begin at

the null character. Otherwise, insertions begin at the position pointed to by str. You can use the ostream::seekp() function to reposition the put pointer.

AIX Considerations

This function is available for 32-bit applications. The second argument is an int value.

Overload 2

```
public:ostrstream(unsigned char* str, long size, int = ios::out)
```

This constructor specifies that the stream buffer that is attached to the ostrstream object consists of an array that starts at the position pointed to by str with a length of size bytes. If ios::ate or ios::app is set, str points to a null-terminated string and insertions begin at the null character. Otherwise, insertions begin at the position pointed to by str. You can use the ostream::seekp() function to reposition the put pointer.

AIX Considerations

This function is available for 64-bit applications. The second argument is a long value.

Overload 3

```
public:ostrstream(char* str, long size, int = ios::out)
```

This constructor specifies that the stream buffer that is attached to the ostrstream object consists of an array that starts at the position pointed to by str with a length of size bytes. If ios::ate or ios::app is set, str points to a null-terminated string and insertions begin at the null character. Otherwise, insertions begin at the position pointed to by str. You can use the ostream::seekp() function to reposition the put pointer.

AIX Considerations

This function is available for 64-bit applications. The second argument is a long value.

Overload 4

```
public:ostrstream(signed char* str, long size, int = ios::out)
```

This constructor specifies that the stream buffer that is attached to the ostrstream object consists of an array that starts at the position pointed to by str with a length of size bytes. If ios::ate or ios::app is set, str points to a null-terminated string and insertions begin at the null character. Otherwise, insertions begin at the position pointed to by str. You can use the ostream::seekp() function to reposition the put pointer.

AIX Considerations

This function is available for 64-bit applications. The second argument is a long value.

Overload 5

```
public:ostrstream(unsigned char* str, int size, int = ios::out)
```

This constructor specifies that the stream buffer that is attached to the ostrstream object consists of an array that starts at the position pointed to by str with a length of size bytes. If ios::ate or ios::app is

set, str points to a null-terminated string and insertions begin at the null character. Otherwise, insertions begin at the position pointed to by str. You can use the ostream::seekp() function to reposition the put pointer.

AIX Considerations

This function is available for 32-bit applications. The second argument is an int value.

Overload 6

```
public:ostrstream(char* str, int size, int = ios::out)
```

This constructor specifies that the stream buffer that is attached to the ostrstream object consists of an array that starts at the position pointed to by str with a length of size bytes. If ios::ate or ios::app is set, str points to a null-terminated string and insertions begin at the null character. Otherwise, insertions begin at the position pointed to by str. You can use the ostream::seekp() function to reposition the put pointer.

AIX Considerations

This function is available for 32-bit applications. The second argument is an int value.

Overload 7

```
public:ostrstream()
```

This constructor specifies that space is allocated dynamically for the stream buffer that is attached to the ostrstream object.

Stream Buffer Functions

Use these functions to work with the stream buffer.

pcount

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

Overload 1

```
public:int pcount()
```

AIX Considerations

This function returns an int value for 32-bit applications. It is not available for 64-bit applications.

Overload 2

```
public:long pcount()
```

AIX Considerations

This function returns a long value for 64-bit applications. It is not available for 32-bit applications.

str

```
public:char* str()
```

Returns a pointer to the stream buffer attached to the ostrstream and calls freeze() with a nonzero value to prevent the stream buffer from being deleted. If the stream buffer was constructed with an explicit array, the

value returned is a pointer to that array. If the stream buffer was constructed in dynamic mode, str points to the dynamically allocated area.

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

ostrstream - Inherited Member Functions and Data

Inherited Public Functions

ios	
Definition	**Page Number**
virtual ~ios()	143
int bad() const	144
static long bitalloc()	149
void clear(int i = 0)	145
int eof() const	145
int fail() const	145
char fill(char)	145
char fill() const	145
long flags() const	146
long flags(long f)	146
int good() const	145
ios(streambuf*)	143
long& iword(int)	149
int operator !() const	149
operator const void *() const	149
operator void *()	149
int precision() const	146
int precision(int)	146
void *& pword(int)	149
streambuf* rdbuf()	149
int rdstate() const	145
long setf(long)	147
long setf(long setbits, long field)	147
int skip(int i)	147
static void sync_with_stdio()	149
ostream* tie(ostream* s)	150

ios	
Definition	Page Number
ostream* tie()	150
long unsetf(long)	147
int width(int w)	148
int width() const	148
static int xalloc()	151
static long xalloc()	151

strstreambase	
Definition	Page Number
strstreambuf* rdbuf()	253

Inherited Public Data

ios	
Definition	Page Number
static const long adjustfield	143
static const long basefield	144
static const long floatfield	144

Inherited Protected Functions

strstreambase	
Definition	Page Number
~strstreambase()	253
strstreambase()	253
strstreambase(char*, int, char*)	253
strstreambase(char*, long, char*)	253

ios	
Definition	Page Number
void init(streambuf*)	149
ios()	143
void setstate(int b)	145

Inherited Protected Data

ios	
Definition	**Page Number**
`static void ( * stdioflush ) ( )`	151
`int assign_private`	144
`streambuf* bp`	144
`int delbuf`	144
`int isfx_special`	144
`int ispecial`	144
`int osfx_special`	144
`int ospecial`	144
`int state`	144
`char x_fill`	148
`long x_flags`	144
`short x_precision`	148
`ostream* x_tie`	144
`short x_width`	148

stdiobuf

This class is used to mix standard C input and output functions with C++ I/O Stream Library functions. This class is obsolete. New programs should avoid using this class.

Class header file: stdiostream.h

stdiobuf - Hierarchy List

streambuf

stdiobuf

stdiobuf - Member Functions and Data by Group

Constructors & Destructor

Objects of the stdiobuf class can be constructed and destructed.

~stdiobuf

```
public:virtual ~stdiobuf()
```

Destructor for stdiobuf. Frees the spaces allocated by the stdiobuf constructor and flushes the file that this stdiobuf object is associated with.

stdiobuf

```
public:stdiobuf(FILE* f)
```

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

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

Positioning Functions

overflow

```
public:virtual int overflow(int = EOF)
```

Emptys an output buffer. Returns EOF on error, 0 otherwise.

pbackfail

```
public:virtual int pbackfail(int c)
```

Attempts to put back a character.

seekoff

```
public:virtual streampos seekoff(streamoff, ios::seek_dir, int)
```

sync

```
public:virtual int sync()
```

underflow

```
public:virtual int underflow()
```

Fills an input buffer. Returns EOF on error or end of input, 0 otherwise.

Query Functions

stdiofile

```
public:FILE* stdiofile()
```

Returns a pointer to the FILE object that the stdiobuf object is associated with.

stdiobuf - Inherited Member Functions and Data

Inherited Public Functions

streambuf	
Definition	**Page Number**
virtual ~streambuf()	235
void dbp()	238
long in_avail()	236
int in_avail()	236
int optim_in_avail()	236
int optim_sbumpc()	236
int out_waiting()	242
long out_waiting()	242
virtual int overflow(int c = EOF)	242
int pptr_non_null()	239

streambuf	
Definition	**Page Number**
`int sbumpc()`	237
`virtual streampos seekoff( streamoff, ios::seek_dir, int = ios::in|ios::out )`	239
`virtual streampos seekpos( streampos, int = ios::in|ios::out )`	239
`virtual streambuf* setbuf(char* p, long len)`	244
`streambuf* setbuf(char* p, int len, int count)`	244
`streambuf* setbuf(unsigned char* p, long len)`	244
`streambuf* setbuf(unsigned char* p, int len)`	244
`virtual streambuf* setbuf(char* p, int len)`	244
`int sgetc()`	237
`long sgetn(char* s, long n)`	237
`int sgetn(char* s, int n)`	237
`int snextc()`	237
`int sputbackc(char c)`	243
`int sputc(int c)`	243
`long sputn(const char* s, long n)`	243
`int sputn(const char* s, int n)`	243
`void stossc()`	240
`streambuf(char* p, int l, int c)`	235
`streambuf(char* p, long l)`	235
`streambuf()`	235
`streambuf(char* p, int l)`	235
`virtual int xsgetn(char* s, int n)`	238
`virtual long xsgetn(char* s, long n)`	238
`virtual int xsputn(const char* s, int n)`	244
`virtual long xsputn(const char* s, long n)`	244

Inherited Public Data

None

Inherited Protected Functions

streambuf	
Definition	**Page Number**
`int allocate()`	246
`char* base()`	240
`long blen() const`	246
`int blen() const`	246
`virtual int doallocate()`	247
`char* eback()`	240
`char* ebuf()`	240
`char* egptr()`	240
`char* epptr()`	240
`void gbump(long n)`	240
`void gbump(int n)`	240
`char* gptr()`	241
`char* pbase()`	241
`void pbump(long n)`	241
`void pbump(int n)`	241
`char* pptr()`	241
`void setb(char* b, char* eb, int a = 0)`	242
`void setg(char* eb, char* g, char* eg)`	242
`void setp(char* p, char* ep)`	242
`void unbuffered(int unb)`	247
`int unbuffered() const`	247

Inherited Protected Data

None

stdiostream

This class uses stdiobuf objects as stream buffers.

Class header file: stdiostream.h

stdiostream - Hierarchy List

ios
stdiostream

stdiostream - Member Functions and Data by Group

Constructors & Destructor

Objects of the stdiostream class can be constructed and destructed.

~stdiostream

```
public:~stdiostream()
```

Destructs a stdiostream object.

stdiostream

```
public:stdiostream(FILE*)
```

Creates a stdiostream object that is attached to the FILE pointed to by the argument.

Miscellaneous

rdbuf

```
public:stdiobuf* rdbuf()
```

Returns a pointer to the stdiobuf object that is attached to the stdiostream object.

stdiostream - Inherited Member Functions and Data

Inherited Public Functions

ios	
Definition	**Page Number**
virtual ~ios()	143
int bad() const	144
static long bitalloc()	149
void clear(int i = 0)	145
int eof() const	145
int fail() const	145
char fill() const	145
char fill(char)	145
long flags(long f)	146
long flags() const	146
int good() const	145
ios(streambuf*)	143
long& iword(int)	149
int operator !() const	149
operator const void *() const	149
operator void *()	149
int precision() const	146

ios	
Definition	**Page Number**
`int precision(int)`	146
`void *& pword(int)`	149
`streambuf* rdbuf()`	149
`int rdstate() const`	145
`long setf(long setbits, long field)`	147
`long setf(long)`	147
`int skip(int i)`	147
`static void sync_with_stdio()`	149
`ostream* tie(ostream* s)`	150
`ostream* tie()`	150
`long unsetf(long)`	147
`int width(int w)`	148
`int width() const`	148
`static int xalloc()`	151
`static long xalloc()`	151

Inherited Public Data

ios	
Definition	**Page Number**
`static const long adjustfield`	143
`static const long basefield`	144
`static const long floatfield`	144

Inherited Protected Functions

ios	
Definition	**Page Number**
`void init(streambuf*)`	149
`ios()`	143
`void setstate(int b)`	145

Inherited Protected Data

ios	
Definition	**Page Number**
static void (* stdioflush) ( )	151
int assign_private	144
streambuf* bp	144
int delbuf	144
int isfx_special	144
int ispecial	144
int osfx_special	144
int ospecial	144
int state	144
char x_fill	148
long x_flags	144
short x_precision	148
ostream* x_tie	144
short x_width	148

streambuf

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

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

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

Public interface

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

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

Protected interface

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

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

Class header file: iostream.h

streambuf - Hierarchy List

streambuf
stdiobuf
filebuf
strstreambuf

streambuf - Member Functions and Data by Group

Constructors & Destructor

Objects of the streambuf class can be constructed and destructed.

~streambuf

```
public:virtual ~streambuf()
```

The destructor for streambuf calls sync(). If a stream buffer has been set up and ios::alloc is set, sync() deletes the stream buffer.

streambuf

Overload 1

```
public:streambuf(char* p, long l)
```

Constructs an empty stream buffer of length *l* starting at the position pointed to by *p*.
AIX Considerations

This constructor is available for 64-bit applications. The second argument is a long value.

Overload 2

```
public:streambuf(char* p, int l)
```

Constructs an empty stream buffer of length *l* starting at the position pointed to by *p*.
AIX Considerations

This constructor is available for 32-bit applications. The second argument is an int value.

Overload 3

```
public:streambuf(char* p, int l, int c)
```

This constructor is obsolete. It is included for compatibility with the AT&T C++ Language System Release 1.2. Use strstreambuf.

Overload 4

```
public:streambuf()
```

Constructs an empty stream buffer corresponding to an empty sequence. The values returned by base(), eback(), ebuf(), egptr(), epptr(), pptr(), gptr(), and pbase() are initially all zero for this stream buffer.

Extraction Functions

Functions that extract characters from the ultimate producer, determine if characters are waiting to be extracted and handle underflow situations.
in_avail

Returns the number of characters that are available to be extracted from the get area of the stream buffer object. You can extract the number of characters equal to the value that in_avail() returns without causing an error.

Overload 1

```
public:int in_avail()
```

AIX Considerations

This function returns an int value for 32-bit applications. It is not available for 64-bit applications.

Overload 2

```
public:long in_avail()
```

AIX Considerations

This function returns a long value for 64-bit applications. It is not available for 32-bit applications.

optim_in_avail

```
public:int optim_in_avail()
```

Returns true if the current get pointer is less then the end of the get area.
optim_sbumpc

```
public:int optim_sbumpc()
```

Moves the get pointer past one character and returns the character that it moved past.

sbumpc

```
public:int sbumpc()
```

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

sgetc

```
public:int sgetc()
```

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

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

sgetn

Overload 1

```
public:long sgetn(char* s, long n)
```

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

AIX Considerations

This function is available for 64-bit applications. It accepts a long argument.

Overload 2

```
public:int sgetn(char* s, int n)
```

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

AIX Considerations

This function is available for 32-bit applications. It accepts an int argument.

snextc

```
public:int snextc()
```

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

underflow

```
public:virtual int underflow()
```

Takes characters from the ultimate producer and puts them in the get area.

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

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

xsgetn

Overload 1

```
public:virtual int xsgetn(char* s, int n)
```

Similar to sputn.

AIX Considerations

This function is available for 32-bit applications. The second argument is an int value.

Overload 2

```
public:virtual long xsgetn(char* s, long n)
```

Similar to sgetn.

AIX Considerations

This function is available for 64-bit applications. The second argument is a long value.

Get/Put Pointer Functions

dbp

```
public:void dbp()
```

Writes to standard output the values returned by the following functions:

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

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

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

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

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

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

pptr_non_null

```
public:int pptr_non_null()
```

Returns true if the put pointer is not null.

seekoff

```
public:virtual streampos
  seekoff( streamoff,
           ios::seek_dir,
           int = ios::in|ios::out )
```

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

The default definition of seekoff() returns EOF.

If you define your own seekoff() function, it should return EOF if the derived class does not support repositioning. If the class does support repositioning, seekoff() should return the new position of the affected pointer, or EOF if an error occurs.

The first argument is an offset from a position in the ultimate producer or ultimate consumer. The second argument is a position in the ultimate produce or ultimate consumer. It can have the following values:

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

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

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

seekpos

```
public:virtual streampos
  seekpos( streampos,
           int = ios::in|ios::out )
```

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

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

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

stossc

```
public:void stossc()
```

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

base

```
protected:char* base()
```

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

eback

```
protected:char* eback()
```

Returns a pointer to the lower bound of the space available for the get area of the streambuf. The space between the pointer returned by eback() and the pointer returned by gptr() is available for putback.

ebuf

```
protected:char* ebuf()
```

Returns a pointer to the byte after the last byte of the stream buffer.

egptr

```
protected:char* egptr()
```

Returns a pointer to the byte after the last byte of the get area of the streambuf.

epptr

```
protected:char* epptr()
```

Returns a pointer to the byte after the last byte of the put area of the streambuf.

gbump

Overload 1

```
protected:void gbump(long n)
```

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

The beginning of the get area is equal to the value returned by gptr().

AIX Considerations

This function is available for 64-bit applications. It accepts a long argument.

Overload 2

```
protected:void gbump(int n)
```

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

The beginning of the get area is equal to the value returned by gptr().

AIX Considerations

This function is available for 32-bit applications. It accepts an int argument.

gptr

```
protected:char* gptr()
```

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

pbase

```
protected:char* pbase()
```

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

pbump

Overload 1

```
protected:void pbump(long n)
```

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

The beginning of the put area is equal to the value returned by pptr().

AIX Considerations

This function is available for 64-bit applications. It accepts a long argument.

Overload 2

```
protected:void pbump(int n)
```

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

The beginning of the put area is equal to the value returned by pptr().

AIX Considerations

This function is available for 32-bit applications. It accepts an int argument.

pptr

```
protected:char* pptr()
```

Returns a pointer to the beginning of the put area of the streambuf. The put area consists of the space between the pointer returned by pptr() and the pointer returned by epptr().

setb

```
protected:void setb(char* b, char* eb, int a = 0)
```

Sets the beginning of the existing stream buffer (the pointer returned by base()) to the position pointed to by *b*, and sets the end of the stream buffer (the pointer returned by ebuf()) to the position pointed to by *eb*.

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

setg

```
protected:void setg(char* eb, char* g, char* eg)
```

Sets the beginning of the get area of streambuf (the pointer returned by gptr()) to *g*, and sets the end of the get area (the pointer returned by egptr()) to *eg*. setg() also sets the beginning of the area available for putback (the pointer returned by eback()) to *eb*.

setp

```
protected:void setp(char* p, char* ep)
```

Sets the spaces available for the put area. Both the start (pbase()) and the beginning (pptr()) of the put area are set to the value *p*.

Sets the beginning of the put area of the streambuf (the pointer returned by pptr()) to the position pointed to by *p*, and sets the end of the put area (the pointer returned by epptr()) to the position pointed to by *ep*.

Insertion Functions

Functions that insert characters into the ultimate consumer, determine if characters are waiting to be inserted and handle overflow situations.

out_waiting

Returns the number of characters that are in the put area waiting to be sent to the ultimate consumer.

Overload 1

```
public:int out_waiting()
```

AIX Considerations

This function returns an int value for 32-bit applications. It is not available for 64-bit applications.

Overload 2

```
public:long out_waiting()
```

AIX Considerations

This function returns a long value for 64-bit applications. It is not available for 32-bit applications.

overflow

```
public:virtual int overflow(int c = EOF)
```

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

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

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

pbackfail

```
public:virtual int pbackfail(int c)
```

Called when both of the following conditions are true:

- An attempt has been made to put back a character.
- There is no room in the putback area. The pointer returned by eback() equals the pointer returned by gptr().

The default definition of pbackfail() returns EOF.

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

sputbackc

```
public:int sputbackc(char c)
```

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

sputc

```
public:int sputc(int c)
```

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

sputn

Overload 1

```
public:int sputn(const char* s, int n)
```

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

AIX Considerations

This function is available for 32-bit applications. It accepts an int argument.

Overload 2

```
public:long sputn(const char* s, long n)
```

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

AIX Considerations

This function is available for 64-bit applications. It accepts a long argument.

xsputn

Overload 1

```
public:virtual int xsputn(const char* s, int n)
```

Similar to sputn.

AIX Considerations

This function is available for use when building 32-bit applications. The second argument is an int value.

Overload 2

```
public:virtual long xsputn(const char* s, long n)
```

Similar to sputn.

AIX Considerations

This function is available for use when building 64-bit applications. The second argument is a long value.

Stream Buffer Functions

Functions that work with the underlying streambuf object.

setbuf

Overload 1

```
public:streambuf* setbuf(unsigned char* p, long len)
```

Sets up a stream buffer consisting of the array of bytes starting at *p* with length *len*.

This function is different from setb(). setb() sets pointers to an existing stream buffer. setbuf(), however, creates the stream buffer.

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

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

AIX Considerations

This function is available for 64-bit applications. It accepts an long argument.

Overload 2

```
public:virtual streambuf* setbuf(char* p, long len)
```

Sets up a stream buffer consisting of the array of bytes starting at *p* with length *len*.

This function is different from setb(). setb() sets pointers to an existing stream buffer. setbuf(), however, creates the stream buffer.

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

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

AIX Considerations

This function is available for 64-bit applications. It accepts an long argument.

Overload 3

```
public:streambuf* setbuf(char* p, int len, int count)
```

This function is obsolete. The I/O Stream Library includes it to be compatible with AT&T C++ Language System Release 1.2

AIX Considerations

This function is available for 32-bit applications. It accepts an int argument.

Overload 4

```
public:virtual streambuf* setbuf(char* p, int len)
```

Sets up a stream buffer consisting of the array of bytes starting at *p* with length *len*.

This function is different from setb(). setb() sets pointers to an existing stream buffer. setbuf(), however, creates the stream buffer.

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

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

AIX Considerations

This function is available for 32-bit applications. It accepts an int argument.

Overload 5

```
public:streambuf* setbuf(unsigned char* p, int len)
```

Sets up a stream buffer consisting of the array of bytes starting at *p* with length *len*.

This function is different from setb(). setb() sets pointers to an existing stream buffer. setbuf(), however, creates the stream buffer.

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

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

AIX Considerations

This function is available for 32-bit applications. It accepts an int argument.

sync

```
public:virtual int sync()
```

Synchronizes the stream buffer with the ultimate producer or the ultimate consumer.

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

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

Otherwise, sync() returns EOF.

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

allocate

```
protected:int allocate()
```

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

- 0, if the streambuf has a stream buffer set up (that is, base() returns a nonzero value), or if unbuffered() returns a nonzero value. allocate() does not do any further processing if it returns 0.
- 1, if allocate() does set up a stream buffer.
- EOF, if the attempt to allocate space for the stream buffer fails.

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

blen

Returns the length (in bytes) of the stream buffer.

Overload 1

```
protected:long blen() const
```

AIX Considerations

The value returned is a long when building 64-bit aplications. This function is not available for 32-bit applications.

Overload 2

```
protected:int blen() const
```

AIX Considerations

The value returned is an int when building 32-bit applications. This function is not available for 64-bit applications.

doallocate

```
protected:virtual int doallocate()
```

Called when allocate() determines that space is needed for a stream buffer.

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

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

In your own version of doallocate(), you provide the size of the buffer for your constructor. Assign the buffer size you want to a variable using a #define statement. This variable can then be used in the constructor for your doallocate() function to define the size of the buffer.

unbuffered

Overload 1

```
protected:void unbuffered(int unb)
```

Manipulates the private streambuf variable called the buffering state. If the buffering state is nonzero, a call to allocate() does not set up a stream buffer.

Changes the value of the buffering state to *unb*.

Overload 2

```
protected:int unbuffered() const
```

Manipulates the private streambuf variable called the buffering state. If the buffering state is nonzero, a call to allocate() does not set up a stream buffer.

Returns the current value of the buffering state.

streambuf - Inherited Member Functions and Data

Inherited Public Functions

None

Inherited Public Data

None

Inherited Protected Functions

None

Inherited Protected Data

None

strstream

strstream is the class that specializes iostream to use a strstreambuf for input and output with arrays of characters in memory. You can create an strstream object by associating the object with a previously allocated array of characters. You can then write output to it, read input from it, and apply other operations to it just as you would to another type of stream.

Class header file: strstream.h

strstream - Hierarchy List

ios

strstreambase

strstream

strstream - Member Functions and Data by Group

Constructors & Destructor

Objects of the strstream class can be constructed and destructed.

~strstream

```
public:~strstream()
```

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

strstream

Overload 1

```
public:strstream(char* str, long size, int mode)
```

This constructor specifies that characters should be extracted and inserted into the array of bytes that starts at the position pointed to by str with a length of size bytes. If ios::ate or ios::app is set in mode, str points to a null-terminated string and insertions begin at the null character. Otherwise, insertions begin at the position pointed to by str. You can use the istream::seekg() function to reposition the get pointer anywhere in this array.

AIX Considerations

This function is available for 64-bit applications. The second argument is a long value.

Overload 2

```
public:strstream(char* str, int size, int mode)
```

This constructor specifies that characters should be extracted and inserted into the array of bytes that starts at the position pointed to by str with a length of size bytes. If ios::ate or ios::app is set in mode, str points to a null-terminated string and insertions begin at the null character. Otherwise, insertions begin at the position pointed to by str. You can use the istream::seekg() function to reposition the get pointer anywhere in this array.

AIX Considerations

This function is available for 32-bit applications. The second argument is an int value.

Overload 3

```
public:strstream(signed char* str, long size, int mode)
```

This constructor specifies that characters should be extracted and inserted into the array of bytes that starts at the position pointed to by str with a length of size bytes. If ios::ate or ios::app is set in mode, str points to a null-terminated string and insertions begin at the null character. Otherwise, insertions begin at the position pointed to by str. You can use the istream::seekg() function to reposition the get pointer anywhere in this array.

AIX Considerations

This function is available for 64-bit applications. The second argument is a long value.

Overload 4

```
public:strstream(unsigned char* str, int size, int mode)
```

This constructor specifies that characters should be extracted and inserted into the array of bytes that starts at the position pointed to by str with a length of size bytes. If ios::ate or ios::app is set in mode, str points to a null-terminated string and insertions begin at the null character. Otherwise, insertions begin at the position pointed to by str. You can use the istream::seekg() function to reposition the get pointer anywhere in this array.

AIX Considerations

This function is available for 32-bit applications. The second argument is an int value.

Overload 5

```
public:strstream(signed char* str, int size, int mode)
```

This constructor specifies that characters should be extracted and inserted into the array of bytes that starts at the position pointed to by str with a length of size bytes. If ios::ate or ios::app is set in mode, str points to a null-terminated string and insertions begin at the null character. Otherwise, insertions begin at the position pointed to by str. You can use the istream::seekg() function to reposition the get pointer anywhere in this array.

AIX Considerations

This function is available for 32-bit applications. The second argument is an int value.

Overload 6

```
public:strstream(unsigned char* str, long size, int mode)
```

This constructor specifies that characters should be extracted and inserted into the array of bytes that starts at the position pointed to by str with a length of size bytes. If ios::ate or ios::app is set in mode, str points to a null-terminated string and insertions begin at the null character. Otherwise, insertions begin at the position pointed to by str. You can use the istream::seekg() function to reposition the get pointer anywhere in this array.

AIX Considerations

This function is available for 64-bit applications. The second argument is a long value.

Overload 7

```
public:strstream()
```

This constructor takes no arguments and specifies that space is allocated dynamically for the stream buffer that is attached to the strstream object.

Stream Buffer Functions

str

```
public:char* str()
```

Returns a pointer to the stream buffer attached to the strstream and calls freeze() with a nonzero value to prevent the stream buffer from being deleted. If the stream buffer was constructed with an explicit array, the value returned is a pointer to that array. If the stream buffer was constructed in dynamic mode, str points to the dynamically allocated area.

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

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

strstream - Inherited Member Functions and Data

Inherited Public Functions

ios	
Definition	**Page Number**
`virtual ~ios()`	143
`int bad() const`	144
`static long bitalloc()`	149
`void clear(int i = 0)`	145
`int eof() const`	145
`int fail() const`	145
`char fill() const`	145
`char fill(char)`	145
`long flags(long f)`	146
`long flags() const`	146
`int good() const`	145
`ios(streambuf*)`	143

ios	
Definition	**Page Number**
long& iword(int)	149
int operator !() const	149
operator const void *() const	149
operator void *()	149
int precision() const	146
int precision(int)	146
void *& pword(int)	149
streambuf* rdbuf()	149
int rdstate() const	145
long setf(long setbits, long field)	147
long setf(long)	147
int skip(int i)	147
static void sync_with_stdio()	149
ostream* tie(ostream* s)	150
ostream* tie()	150
long unsetf(long)	147
int width() const	148
int width(int w)	148
static int xalloc()	151
static long xalloc()	151

strstreambase	
Definition	**Page Number**
strstreambuf* rdbuf()	253

Inherited Public Data

ios	
Definition	**Page Number**
static const long adjustfield	143
static const long basefield	144
static const long floatfield	144

Inherited Protected Functions

strstreambase	
Definition	**Page Number**
~strstreambase()	253
strstreambase(char*, int, char*)	253
strstreambase()	253
strstreambase(char*, long, char*)	253

ios	
Definition	**Page Number**
void init(streambuf*)	149
ios()	143
void setstate(int b)	145

Inherited Protected Data

ios	
Definition	**Page Number**
static void (* stdioflush) ( )	151
int assign_private	144
streambuf* bp	144
int delbuf	144
int isfx_special	144
int ispecial	144
int osfx_special	144
int ospecial	144
int state	144
char x_fill	148
long x_flags	144
short x_precision	148
ostream* x_tie	144
short x_width	148

strstreambase

The strstreambase class is an internal class that provides common functions for the classes that are derived from it; strstream, istrstream, and ostrstream. Do not use the strstreambase class directly.

Class header file: strstream.h

strstreambase - Hierarchy List

ios
strstreambase
strstream
istrstream
ostrstream

strstreambase - Member Functions and Data by Group

Constructors & Destructor

Objects of the strstreambase class can be constructed and destructed by objects derived from it. Do not use these functions directly.

~strstreambase

```
protected:~strstreambase()
```

Destructs a strstreambase object.

strstreambase

Overload 1

```
protected:strstreambase(char*, long, char*)
```

Constructs a strstreambase object.

AIX Considerations

This function is available for 64-bit applications. The second argument is a long value.

Overload 2

```
protected:strstreambase(char*, int, char*)
```

Constructs a strstreambase object.

AIX Considerations

This function is available for 32-bit applications. The second argument is an int value.

Overload 3

```
protected:strstreambase()
```

Constructs a strstreambase object.

Misc

rdbuf

```
public:strstreambuf* rdbuf()
```

Returns a pointer to the stream buffer that the strstreambase object is attached to.

strstreambase - Inherited Member Functions and Data

Inherited Public Functions

ios	
Definition	**Page Number**
virtual ~ios()	143
int bad() const	144
static long bitalloc()	149
void clear(int i = 0)	145
int eof() const	145
int fail() const	145
char fill() const	145
char fill(char)	145
long flags(long f)	146
long flags() const	146
int good() const	145
ios(streambuf*)	143
long& iword(int)	149
int operator !() const	149
operator const void *() const	149
operator void *()	149
int precision() const	146
int precision(int)	146
void *& pword(int)	149
streambuf* rdbuf()	149
int rdstate() const	145
long setf(long setbits, long field)	147
long setf(long)	147
int skip(int i)	147
static void sync_with_stdio()	149
ostream* tie(ostream* s)	150
ostream* tie()	150
long unsetf(long)	147
int width() const	148
int width(int w)	148
static int xalloc()	151

ios	
Definition	**Page Number**
`static long xalloc()`	151

Inherited Public Data

ios	
Definition	**Page Number**
`static const long adjustfield`	143
`static const long basefield`	144
`static const long floatfield`	144

Inherited Protected Functions

ios	
Definition	**Page Number**
`void init(streambuf*)`	149
`ios()`	143
`void setstate(int b)`	145

Inherited Protected Data

ios	
Definition	**Page Number**
`static void ( * stdioflush ) ( )`	151
`int assign_private`	144
`streambuf* bp`	144
`int delbuf`	144
`int isfx_special`	144
`int ispecial`	144
`int osfx_special`	144
`int ospecial`	144
`int state`	144
`char x_fill`	148
`long x_flags`	144
`short x_precision`	148
`ostream* x_tie`	144
`short x_width`	148

strstreambuf

This class specializes streambuf to use an array of bytes in memory as the source or target of data.

Class header file: strstream.h

strstreambuf - Hierarchy List

streambuf

strstreambuf

strstreambuf - Member Functions and Data by Group

Constructors & Destructor

Objects of the strstreambuf class can be constructed and destructed.

~strstreambuf

```
public:~strstreambuf()
```

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

- If you created the strstreambuf object using the constructor that takes two pointers to functions as arguments, the destructor frees the space allocated by the destructor by calling the function pointed to by the second argument to the constructor.
- If you created the strstreambuf object using any of the other constructors, the destructor calls the delete operator to free the space allocated by the constructor.

strstreambuf

Overload 1

```
public:strstreambuf(long)
```

This constructor takes one argument and constructs an empty strstreambuf object in dynamic mode. The initial size of the stream buffer will be at least as long as the argument in bytes.

AIX Considerations

This constructor is available for 64-bit applications. It accepts a long argument.

Overload 2

```
public:strstreambuf(int)
```

This constructor takes one argument and constructs an empty strstreambuf object in dynamic mode. The initial size of the stream buffer will be at least as long as the argument in bytes.

AIX Considerations

This constructor is available for 32-bit applications. It accepts an int argument.

Overload 3

```
public:strstreambuf(char* b, int size, char* pstart = 0)
```

Constructs a strstreambuf object with a stream buffer that begins at the position pointed to by *b*. The nature of the stream buffer depends on the value of *size*.

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

Regardless of the values of *size*, if the value of *pstart* is 0, the get area will include the entire stream buffer, and insertions will caues errors.

AIX Considerations

This constructor is available for 32-bit applications. The second argument is an int value.

Overload 4

```
public:strstreambuf( unsigned char* b,
            int size,
            unsigned char* pstart = 0 )
```

Constructs a strstreambuf object with a stream buffer that begins at the position pointed to by *b*. The nature of the stream buffer depends on the value of *size*.

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

Regardless of the values of *size*, if the value of *pstart* is 0, the get area will include the entire stream buffer, and insertions will caues errors.

AIX Considerations

This constructor is available for 32-bit applications. The second argument is an int value.

Overload 5

```
public:strstreambuf( unsigned char* b,
            long size,
            unsigned char* pstart = 0 )
```

Constructs a strstreambuf object with a stream buffer that begins at the position pointed to by *b*. The nature of the stream buffer depends on the value of *size*.

- If *size* is positive, the *size* bytes following the position pointed to by *b* make up the stream buffer.

- If *size* equals 0, *b* points to the beginning of a null-terminated string, and the bytes of that string, excluding the terminating null character, will make up the stream buffer.
- If *size* is negative, the stream buffer has an indefinite length. The get pointer of the stream buffer is initialized to *b*, and the put pointer is initialized to *pstart*.

Regardless of the values of *size*, if the value of *pstart* is 0, the get area will include the entire stream buffer, and insertions will caues errors.

AIX Considerations

This constructor is available for 64-bit applications. The second argument is a long value.

Overload 6

```
public:strstreambuf( void * ( * a ) ( long ),
             void ( * f ) ( void * ) )
```

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

Overload 7

```
public:strstreambuf( signed char* b,
             int size,
             signed char* pstart = 0 )
```

Constructs a strstreambuf object with a stream buffer that begins at the position pointed to by *b*. The nature of the stream buffer depends on the value of *size*.

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

Regardless of the values of *size*, if the value of *pstart* is 0, the get area will include the entire stream buffer, and insertions will caues errors.

AIX Considerations

This constructor is available for 32-bit applications. The second argument is an int value.

Overload 8

```
public:strstreambuf( signed char* b,
             long size,
             signed char* pstart = 0 )
```

Constructs a strstreambuf object with a stream buffer that begins at the position pointed to by *b*. The nature of the stream buffer depends on the value of *size*.

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

Regardless of the values of *size*, if the value of *pstart* is 0, the get area will include the entire stream buffer, and insertions will caues errors.

AIX Considerations

This constructor is available for 64-bit applications. The second argument is a long value.

Overload 9

```
public:strstreambuf(char* b, long size, char* pstart = 0)
```

Constructs a strstreambuf object with a stream buffer that begins at the position pointed to by *b*. The nature of the stream buffer depends on the value of *size*.

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

Regardless of the values of *size*, if the value of *pstart* is 0, the get area will include the entire stream buffer, and insertions will caues errors.

AIX Considerations

This constructor is available for 64-bit applications. The second argument is a long value.

Overload 10

```
public:strstreambuf()
```

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

Get/Put Pointer Functions

seekoff

```
public:virtual streampos seekoff(streamoff, ios::seek_dir, int)
```

Repositions the get or put pointer in the array of bytes in memory that serves as the ultimate producer or consumer.

If you constructed the strstreambuf in dynamic mode, the results of seekoff() are unpredictable. Therefore, do not use seekoff() with an strstreambuf object that you created in dynamic mode.

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

seekoff() attempts to reposition the affected pointer to the value of ios::seek_dir + streamoff. ios::seek_dir can have the following values: ios::beg, ios::cur, or ios::end.

If the value of ios::seek_dir + streamoff is equal to or greater than the end of the array, the value is not valid and seekoff() returns EOF. Otherwise, seekoff() sets the affected pointer to this value and returns this value.

Insertion & Extraction Functions

overflow

```
public:virtual int overflow(int)
```

Causes the ultimate consumer to consume the characters in the put area and calls setp() to establish a new put area. The argument is stored in the new put area if its value is not equal to EOF.

pcount

This function is internal and should not be used.

Overload 1

```
public:long pcount()
```

AIX Considerations

This function returns a long for 64-bit applications. It is not available for 32-bit applications.

Overload 2

```
public:int pcount()
```

AIX Considerations

This function returns an int for 32-bit applications. It is not available for 64-bit applications.

underflow

```
public:virtual int underflow()
```

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

Stream Buffer Functions

doallocate

```
public:virtual int doallocate()
```

Attempts to allocate space for a stream buffer. If you created the strstreambuf object using the constructor that takes two pointers to functions as arguments, doallocate() allocates space for the stream buffer by calling the function pointed to by the first argument to the constructor. Otherwise, doallocate() calls the operator new to allocate space for the stream buffer.

freeze

```
public:void freeze(int n = 1)
```

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

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

isfrozen

```
public:int isfrozen()
```

Returns true if the stream buffer is frozen.

setbuf

Overload 1

```
public:virtual streambuf* setbuf(char* p, long l)
```

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

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

AIX Considerations

This function is available for 64-bit applications. The second argument is a long value.

Overload 2

```
public:virtual streambuf* setbuf(char* p, int l)
```

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

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

AIX Considerations

This function is available for 32-bit applications. The second argument is an int value.

str

```
public:char* str()
```

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

strstreambuf - Inherited Member Functions and Data

Inherited Public Functions

streambuf	
Definition	**Page Number**
`virtual ~streambuf()`	235
`void dbp()`	238
`long in_avail()`	236
`int in_avail()`	236
`int optim_in_avail()`	236
`int optim_sbumpc()`	236
`long out_waiting()`	242
`int out_waiting()`	242
`virtual int overflow(int c = EOF)`	242
`virtual int pbackfail(int c)`	243
`int pptr_non_null()`	239
`int sbumpc()`	237
`virtual streampos seekoff( streamoff, ios::seek_dir, int = ios::in|ios::out )`	239
`virtual streampos seekpos( streampos, int = ios::in|ios::out )`	239
`streambuf* setbuf(unsigned char* p, long len)`	244
`virtual streambuf* setbuf(char* p, long len)`	244
`virtual streambuf* setbuf(char* p, int len)`	244
`streambuf* setbuf(unsigned char* p, int len)`	244
`streambuf* setbuf(char* p, int len, int count)`	244
`int sgetc()`	237

streambuf	
Definition	**Page Number**
int sgetn(char* s, int n)	237
long sgetn(char* s, long n)	237
int snextc()	237
int sputbackc(char c)	243
int sputc(int c)	243
int sputn(const char* s, int n)	243
long sputn(const char* s, long n)	243
void stossc()	240
streambuf(char* p, long l)	235
streambuf(char* p, int l, int c)	235
streambuf()	235
streambuf(char* p, int l)	235
virtual int sync()	246
virtual int xsgetn(char* s, int n)	238
virtual long xsgetn(char* s, long n)	238
virtual int xsputn(const char* s, int n)	244
virtual long xsputn(const char* s, long n)	244

Inherited Public Data

None

Inherited Protected Functions

streambuf	
Definition	**Page Number**
int allocate()	246
char* base()	240
int blen() const	246
long blen() const	246
char* eback()	240
char* ebuf()	240
char* egptr()	240
char* epptr()	240
void gbump(long n)	240
void gbump(int n)	240

streambuf	
Definition	**Page Number**
`char* gptr()`	241
`char* pbase()`	241
`void pbump(int n)`	241
`void pbump(long n)`	241
`char* pptr()`	241
`void setb(char* b, char* eb, int a = 0)`	242
`void setg(char* eb, char* g, char* eg)`	242
`void setp(char* p, char* ep)`	242
`int unbuffered() const`	247
`void unbuffered(int unb)`	247

Inherited Protected Data

None

Appendix. MEMDBG Library Functions

_debug_calloc — Allocate and Initialize Memory

Format

```
#include <stdlib.h>   /* also in <malloc.h> */
void *_debug_calloc(size_t num, size_t size,
                    const char *file, size_t line);
```

Language Level: Extension
_debug_calloc is the debug version of calloc. Like calloc, it allocates memory from the default heap for an array of *num* elements, each of length *size* bytes. It then initializes all bits of each element to 0.

In addition, _debug_calloc makes an implicit call to _heap_check, and stores the name of the file *file* and the line number *line* where the storage is allocated. This information can be used later by the _heap_check, _dump_allocated or _dump_allocated_delta functions.

To use _debug_calloc, you must compile with the debug memory -qheapdebug option. This option maps all calloc calls to _debug_calloc.

Note: -qheapdebug maps all calls to memory management functions (including heap-specific versions) to their debug counterparts. To prevent a call from being mapped, parenthesize the function name.

To reallocate or free memory allocated by _debug_calloc, use _debug_realloc and _debug_free; you can also use realloc and free if you do not want debug information about the operation.

A heap-specific version (_debug_ucalloc) is available. _debug_calloc always allocates memory from the default heap.

Return Value
_debug_calloc returns a pointer to the reserved space. If not enough memory is available, or if *num* or *size* is 0, _debug_calloc returns NULL.

Example
This example reserves storage of 100 bytes. It then attempts to write to storage that was not allocated. When _debug_calloc is called again, _heap_check detects the error, generates several messages, and stops the program.

Note: You must compile this example with -qheapdebug to map the calloc calls to _debug_calloc.

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

int main(void)
{
   char *ptr1, *ptr2;

   if (NULL == (ptr1 = (char*)calloc(1, 100))) {
      puts("Could not allocate memory block.");
      exit(EXIT_FAILURE);
   }
```

© Copyright IBM Corp. 1998, 2002

```
   memset(ptr1, 'a', 105);         /* overwrites storage that was not allocated */
   ptr2 = (char*)calloc(2, 20);      /* this call to calloc invokes _heap_check */
   puts("_debug_calloc did not detect that a memory block was overwritten.");
   return 0;

   /****************************************************************************
      The output should be similar to:

      End of allocated object 0x00073890 was overwritten at 0x000738f4.
      The first eight bytes of the memory block (in hex) are: 6161616161616161.
      This memory block was (re)allocated at line number 9 in _debug_callo.c.
      Heap state was valid at line 9 of _debug_callo.c.
      Memory error detected at line 14 of _debug_callo.c.
   ****************************************************************************/
}
```

RELATED CONCEPTS

Debugging Memory Heaps (page 29)
Memory Management Functions (page 25)
Managing Memory with Multiple Memory Heaps (page 27)

RELATED TASKS

Debug Programs with Heap Memory (page 35)

RELATED REFERENCES

_debug_free - Free Allocated Memory (page 266)
_debug_heapmin - Free Unused Memory in the Default Heap (page 268)
_debug_malloc - Allocate Memory (page 269)
_debug_memcpy - Copy Bytes (page 271)
_debug_memmove - Copy Bytes (page 273)
_debug_memset - Set Bytes to Value (page 274)
_debug_realloc - Reallocate Memory Block (page 276)
_debug_strcat - Concatenate Strings (page 278)
_debug_strcpy - Copy Strings (page 279)
_debug_strncat - Concatenate Strings (page 282)
_debug_strncpy - Copy Strings (page 284)
_debug_strnset - Set Characters in String (page 281)
_debug_strset - Set Characters in String (page 285)
_debug_ucalloc - Reserve and Initialize Memory from User Heap (page 287)
_debug_uheapmin - Free Unused Memory in User Heap (page 289)
_debug_umalloc - Reserve Memory Block from User Heap (page 290)
-qheapdebug Compile Option

_debug_free — Free Allocated Memory

Format

```
#include <stdlib.h>    /* also in <malloc.h> */
void _debug_free(void *ptr, const char *file,
                        size_t line);
```

Language Level: Extension
_debug_free is the debug version of free. Like free, it frees the block of memory pointed to by *ptr*. _debug_free also sets each block of freed memory to 0xFB, so you can easily locate instances where your program uses the data in freed memory.

In addition, _debug_free makes an implicit call to the _heap_check, and stores the file name *file* and the line number *line* where the memory is freed. This information can be used later by the _heap_check, _dump_allocated, or _dump_allocated_delta functions.

To use _debug_free, you must compile with the -qheapdebug option. This option maps all free calls to _debug_free.

Note: -qheapdebug maps all calls to memory management functions (including heap-specific versions) to their debug counterparts. To prevent a call from being mapped, parenthesize the function name.

Because _debug_free always checks what heap the memory was allocated from, you can use _debug_free to free memory blocks allocated by the regular, heap-specific, or debug versions of the memory management functions. However, if the memory was not allocated by the memory management functions, or was previously freed, _debug_free generates an error message and the program ends.

Return Value
There is no return value.

Example
This example reserves two blocks, one of 10 bytes and the other of 20 bytes. It then frees the first block and attempts to overwrite the freed storage. When _debug_free is called a second time, _heap_check detects the error, prints out several messages, and stops the program.

Note: You must compile this example with -qheapdebug to map the free calls to _debug_free.

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

int main(void)
{
   char *ptr1, *ptr2;

   if (NULL == (ptr1 = (char*)malloc(10)) || NULL == (ptr2 = (char*)malloc(20))) {
      puts("Could not allocate memory block.");
      exit(EXIT_FAILURE);
   }
   free(ptr1);
   memset(ptr1, 'a', 5);      /* overwrites storage that has been freed       */
   free(ptr2);                 /* this call to free invokes _heap_check       */
   puts("_debug_free did not detect that a freed memory block was overwritten.");
   return 0;

   /****************************************************************************
      The output should be similar to:

      Free heap was overwritten at 0x00073890.
      Heap state was valid at line 12 of _debug_free.c.
      Memory error detected at line 14 of _debug_free.c.
   ****************************************************************************/
}
```

RELATED CONCEPTS

Debugging Memory Heaps (page 29)
Memory Management Functions (page 25)
Managing Memory with Multiple Memory Heaps (page 27)

RELATED TASKS

Debug Programs with Heap Memory (page 35)

RELATED REFERENCES

_debug_calloc - Allocate and Initialize Memory (page 265)
_debug_heapmin - Free Unused Memory in the Default Heap (page 268)

_debug_malloc - Allocate Memory (page 269)
_debug_memcpy - Copy Bytes (page 271)
_debug_memmove - Copy Bytes (page 273)
_debug_memset - Set Bytes to Value (page 274)
_debug_realloc - Reallocate Memory Block (page 276)
_debug_strcat - Concatenate Strings (page 278)
_debug_strcpy - Copy Strings (page 279)
_debug_strncat - Concatenate Strings (page 282)
_debug_strncpy - Copy Strings (page 284)
_debug_strnset - Set Characters in String (page 281)
_debug_strset - Set Characters in String (page 285)
_debug_ucalloc - Reserve and Initialize Memory from User Heap (page 287)
_debug_uheapmin - Free Unused Memory in User Heap (page 289)
_debug_umalloc - Reserve Memory Block from User Heap (page 290)
-qheapdebug Compile Option

_debug_heapmin — Free Unused Memory in the Default Heap

Format

```
#include <stdlib.h>  /* also in <malloc.h> */
int _debug_heapmin(const char *file, size_t line);
```

Language Level: Extension
_debug_heapmin is the debug version of _heapmin. Like _heapmin, it returns all unused memory from the default runtime heap to the operating system.

In addition, _debug_heapmin makes an implicit call to _heap_check, and stores the file name *file* and the line number *line* where the memory is returned. This information can be used later by the _heap_check function.

To use _debug_heapmin, you must compile with the -qheapdebug option. This option maps all _heapmin calls to _debug_heapmin.

Note: -qheapdebug maps all calls to memory management functions (including heap-specific versions) to their debug counterparts. To prevent a call from being mapped, parenthesize the function name.

A heap-specific version of this function (_debug_uheapmin) is also available. _debug_heapmin always operates on the default heap.

Return Value
If successful, _debug_heapmin returns 0; otherwise, it returns -1.

Example
This example allocates 10000 bytes of storage, changes the storage size to 10 bytes, and then uses _debug_heapmin to return the unused memory to the operating system. The program then attempts to overwrite memory that was not allocated. When _debug_heapmin is called again, _heap_check detects the error, generates several messages, and stops the program.

Note: You must compile this example with the -qheapdebug option to map the _heapmin calls to _debug_heapmin.

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

```
int main(void)
{
   char *ptr;

   /* Allocate a large object from the system */
   if (NULL == (ptr = (char*)malloc(100000))) {
      puts("Could not allocate memory block.");
      exit(EXIT_FAILURE);
   }
   ptr = (char*)realloc(ptr, 10);
   _heapmin();                /* No allocation problems to detect            */

   *(ptr - 1) = 'a';          /* Overwrite memory that was not allocated     */
   _heapmin();                /* This call to _heapmin invokes _heap_check   */

   puts("_debug_heapmin did not detect that a non-allocated memory block"
        "was overwritten.");
   return 0;

   /****************************************************************************
      Possible output is:

      Header information of object 0x000738b0 was overwritten at 0x000738ac.
      The first eight bytes of the memory block (in hex) are: AAAAAAAAAAAAAAAA.
      This memory block was (re)allocated at line number 13 in _debug_heapm.c.
      Heap state was valid at line 14 of _debug_heapm.c.
      Memory error detected at line 17 of _debug_heapm.c.
   ****************************************************************************/
}
```

RELATED CONCEPTS

Debugging Memory Heaps (page 29)
Memory Management Functions (page 25)
Managing Memory with Multiple Memory Heaps (page 27)

RELATED TASKS

Debug Programs with Heap Memory (page 35)

RELATED REFERENCES

_debug_calloc - Allocate and Initialize Memory (page 265)
_debug_free - Free Allocated Memory (page 266)
_debug_malloc - Allocate Memory (page 269)
_debug_memcpy - Copy Bytes (page 271)
_debug_memmove - Copy Bytes (page 273)
_debug_memset - Set Bytes to Value (page 274)
_debug_realloc - Reallocate Memory Block (page 276)
_debug_strcat - Concatenate Strings (page 278)
_debug_strcpy - Copy Strings (page 279)
_debug_strncat - Concatenate Strings (page 282)
_debug_strncpy - Copy Strings (page 284)
_debug_strnset - Set Characters in String (page 281)
_debug_strset - Set Characters in String (page 285)
_debug_ucalloc - Reserve and Initialize Memory from User Heap (page 287)
_debug_uheapmin - Free Unused Memory in User Heap (page 289)
_debug_umalloc - Reserve Memory Block from User Heap (page 290)
-qheapdebug Compile Option

_debug_malloc — Allocate Memory

Format

```
#include <stdlib.h>  /* also in <malloc.h> */
void *_debug_malloc(size_t size,
                    const char *file, size_t line);
```

Language Level: Extension
_debug_malloc is the debug version of malloc. Like malloc, it reserves a block of storage of *size* bytes from the default heap. _debug_malloc also sets all the memory it allocates to 0xAA, so you can easily locate instances where your program uses the data in the memory without initializing it first.

In addition, _debug_malloc makes an implicit call to _heap_check, and stores the file name *file* and the line number *line* where the storage is allocated. This information can later be used by the _heap_check , _dump_allocated, or _dump_allocated_delta functions.

To use _debug_malloc, you must compile with the -qheapdebug option. This option maps all malloc calls to _debug_malloc.

Note: -qheapdebug maps all calls to memory management functions (including heap-specific versions) to their debug counterparts. To prevent a call from being mapped, parenthesize the function name.

To reallocate or free memory allocated by _debug_malloc, use _debug_realloc and _debug_free; you can also use realloc and free if you do not want debug information about the operation.

A heap-specific version of this function (_debug_umalloc) is also available. _debug_malloc always allocates memory from the default heap.

Return Value
_debug_malloc returns a pointer to the reserved space. If not enough memory is available or if *size* is 0, _debug_malloc returns NULL.

Example
This example allocates 100 bytes of storage. It then attempts to write to storage that was not allocated. When _debug_malloc is called again, _heap_check detects the error, generates several messages, and stops the program.

Note: You must compile this example with the -qheapdebug option to map the malloc calls to _debug_malloc.

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

int main(void)
{
   char *ptr1, *ptr2;

   if (NULL == (ptr1 = (char*)malloc(100))) {
      puts("Could not allocate memory block.");
      exit(EXIT_FAILURE);
   }
   *(ptr1 - 1) = 'a';        /* overwrites storage that was not allocated    */
   ptr2 = (char*)malloc(10); /* this call to malloc invokes _heap_check      */
   puts("_debug_malloc did not detect that a memory block was overwritten.");
   return 0;

   /*************************************************************************
      Possible output is:

      Header information of object 0x00073890 was overwritten at 0x0007388c.
      The first eight bytes of the memory block (in hex) are: AAAAAAAAAAAAAAAA.
      This memory block was (re)allocated at line number 8 in _debug_mallo.c.
      Heap state was valid at line 8 of _debug_mallo.c.
      Memory error detected at line 13 of _debug_mallo.c.
   *************************************************************************/
}
```

RELATED CONCEPTS

Debugging Memory Heaps (page 29)
Memory Management Functions (page 25)
Managing Memory with Multiple Memory Heaps (page 27)

RELATED TASKS

Debug Programs with Heap Memory (page 35)

RELATED REFERENCES

_debug_free - Free Allocated Memory (page 266)
_debug_heapmin - Free Unused Memory in the Default Heap (page 268)
_debug_memcpy - Copy Bytes (page 271)
_debug_memmove - Copy Bytes (page 273)
_debug_memset - Set Bytes to Value (page 274)
_debug_realloc - Reallocate Memory Block (page 276)
_debug_strcat - Concatenate Strings (page 278)
_debug_strcpy - Copy Strings (page 279)
_debug_strncat - Concatenate Strings (page 282)
_debug_strncpy - Copy Strings (page 284)
_debug_strnset - Set Characters in String (page 281)
_debug_strset - Set Characters in String (page 285)
_debug_ucalloc - Reserve and Initialize Memory from User Heap (page 287)
_debug_uheapmin - Free Unused Memory in User Heap (page 289)
_debug_umalloc - Reserve Memory Block from User Heap (page 290)
-qheapdebug Compile Option

_debug_memcpy — Copy Bytes

Format

```
#include <string.h>
void *_debug_memcpy(void *dest, const void *src, size_t count,
                    const char *file, size_t line);
```

Language Level: Extension

_debug_memcpy is the debug version of memcpy. Like memcpy, it copies *count* bytes of *src* to *dest*, where the behavior is undefined if copying takes place between objects that overlap.

_debug_memcpy validates the heap after copying the bytes to the target location, and performs this check only when the target is within a heap. _debug_memcpy makes an implicit call to _heap_check. If _debug_memcpy detects a corrupted heap when it makes a call to _heap_check, _debug_memcpy will report the file name *file* and line number *line* in a message.

Note: _debug_memcpy checks only the current default heap. Therefore, this debug support will not check all heaps within applications that have multiple user heaps.

To use _debug_memcpy, you must compile with the -qheapdebug option. This option maps all memcpy calls to _debug_memcpy. You do not have to change your source code, in order for _debug_memcpy to verify the heap.

Note: -qheapdebug maps all calls to other string functions and all calls to memory management functions (including a heap-specific version), to their debug counterparts. To prevent a call from being mapped, parenthesize the function name.

Return Value
_debug_memcpy returns a pointer to *dest*.

Example
This example contains a programming error. On the memcpy used to initialize the target location, the count is more than the size of the target object, and the memcpy operation copies bytes past the end of the allocated object.

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

#define  MAX_LEN      10

int main(void)
{
   char *source, *target;

   target = (char*)malloc(MAX_LEN);
   memcpy(target, "This is the target string", 11);

   printf("Target is \"%s\"\n", target);
   return 0;

   /****************************************************************************
      The output should be similar to:

      End of allocated object 0x00073c80 was overwritten at 0x00073c8a.
      The first eight bytes of the memory block (in hex) are: 5468697320697320.
      This memory block was (re)allocated at line number 11 in memcpy.c.
      Heap state was valid at line 11 of memcpy.c.
      Memory error detected at line 12 of memcpy.c.
   ****************************************************************************/
}
```

RELATED CONCEPTS

Debugging Memory Heaps (page 29)
Memory Management Functions (page 25)
Managing Memory with Multiple Memory Heaps (page 27)

RELATED TASKS

Debug Programs with Heap Memory (page 35)

RELATED REFERENCES

_debug_calloc - Allocate and Initialize Memory (page 265)
_debug_free - Free Allocated Memory (page 266)
_debug_heapmin - Free Unused Memory in the Default Heap (page 268)
_debug_malloc - Allocate Memory (page 269)
_debug_memmove - Copy Bytes (page 273)
_debug_memset - Set Bytes to Value (page 274)
_debug_realloc - Reallocate Memory Block (page 276)
_debug_strcat - Concatenate Strings (page 278)
_debug_strcpy - Copy Strings (page 279)
_debug_strncat - Concatenate Strings (page 282)
_debug_strncpy - Copy Strings (page 284)
_debug_strnset - Set Characters in String (page 281)
_debug_strset - Set Characters in String (page 285)
_debug_ucalloc - Reserve and Initialize Memory from User Heap (page 287)
_debug_uheapmin - Free Unused Memory in User Heap (page 289)
_debug_umalloc - Reserve Memory Block from User Heap (page 290)
-qheapdebug Compile Option

_debug_memmove — Copy Bytes

Format

```
#include <string.h>
void *_debug_memmove(void *dest, const void *src, size_t count,
                     const char *file, size_t line);
```

Language Level: Extension
_debug_memmove is the debug version of memmove. Like memmove, it copies *count* bytes of *src* to *dest*, and allows for copying between objects that may overlap.

_debug_memmove validates the heap after copying the bytes to the target location, and performs this check only when the target is within a heap. _debug_memmove makes an implicit call to _heap_check. If _debug_memmove detects a corrupted heap when it makes a call to _heap_check, _debug_memmove will report the file name *file* and line number *line* in a message.

Note: _debug_memmove checks only the current default heap. Therefore, this debug support will not check all heaps within applications that have multiple user heaps.

To use _debug_memmove, you must compile with the -qheapdebug option. This option maps all memcpy calls to _debug_memmove. You do not have to change your source code, in order for _debug_memmove to verify the heap.

Note: -qheapdebug maps all calls to other string functions and all calls to memory management functions (including a heap-specific version), to their debug counterparts. To prevent a call from being mapped, parenthesize the function name.

Return Value
_debug_memmove returns a pointer to *dest*.

Example
This example contains a programming error. The count specified on memmove is 15 instead of 5, and the memmove operation copies bytes past the end of the allocated object.

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

#define  SIZE         21

int main(void)
{
   char *target, *p, *source;

   target = (char*)malloc(SIZE);
   strcpy(target, "a shiny white sphere");
   p = target+8;                  /* p points at the starting character
                                     of the word we want to replace   */
   source = target+2;             /* start of "shiny" */

   printf("Before memmove, target is \"%s\"\n", target);
   memmove(p, source, 15);
   printf("After memmove, target becomes \"%s\"\n", target);
   return 0;

   /****************************************************************************
      The output should be similar to:
```

```
      Before memmove, target is "a shiny white sphere"
      End of allocated object 0x00073c80 was overwritten at 0x00073c95.
      The first eight bytes of the memory block (in hex) are: 61207368696E7920.
      This memory block was (re)allocated at line number 11 in memmove.c.
      Heap state was valid at line 12 of memmove.c.
      Memory error detected at line 18 of memcpy.c.
   *************************************************************************/
}
```

RELATED CONCEPTS

Debugging Memory Heaps (page 29)
Memory Management Functions (page 25)
Managing Memory with Multiple Memory Heaps (page 27)

RELATED TASKS

Debug Programs with Heap Memory (page 35)

RELATED REFERENCES

_debug_calloc - Allocate and Initialize Memory (page 265)
_debug_free - Free Allocated Memory (page 266)
_debug_heapmin - Free Unused Memory in the Default Heap (page 268)
_debug_malloc - Allocate Memory (page 269)
_debug_memcpy - Copy Bytes (page 271)
_debug_memset - Set Bytes to Value (page 274)
_debug_realloc - Reallocate Memory Block (page 276)
_debug_strcat - Concatenate Strings (page 278)
_debug_strcpy - Copy Strings (page 279)
_debug_strncat - Concatenate Strings (page 282)
_debug_strncpy - Copy Strings (page 284)
_debug_strnset - Set Characters in String (page 281)
_debug_strset - Set Characters in String (page 285)
_debug_ucalloc - Reserve and Initialize Memory from User Heap (page 287)
_debug_uheapmin - Free Unused Memory in User Heap (page 289)
_debug_umalloc - Reserve Memory Block from User Heap (page 290)
-qheapdebug Compile Option

_debug_memset — Set Bytes to Value

Format

```
#include <string.h>
void *_debug_memset(void *dest, int c, size_t count,
                    const char *file, size_t line);
```

Language Level: Extension

_debug_memset is the debug version of memset. Like memset, it sets the first *count* bytes of *dest* to the value *c*. The value of *c* is converted to an unsigned character.

_debug_memset validates the heap after setting the bytes, and performs this check only when the target is within a heap. _debug_memset makes an implicit call to _heap_check. If _debug_memset detects a corrupted heap when it makes a call to _heap_check, _debug_memset will report the file name *file* and line number *line* in a message.

Note: _debug_memset checks only the current default heap. Therefore, this debug support will not check all heaps within applications that have multiple user heaps.

To use _debug_memset, you must compile with the -qheapdebug option. This option maps all memset calls to _debug_memset. You do not have to change your source code, in order for _debug_memset to verify the heap.

Note: -qheapdebug maps all calls to other string functions and all calls to memory management functions (including a heap-specific version), to their debug counterparts. To prevent a call from being mapped, parenthesize the function name.

Return Value
_debug_memset returns a pointer to *dest*.

Example
This example contains a programming error. The invocation of memset that puts 'B' in the buffer specifies the wrong count, and stores bytes past the end of the buffer.

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

#define  BUF_SIZE     20

int main(void)
{
   char *buffer, *buffer2;
   char *string;

   buffer = (char*)calloc(1, BUF_SIZE+1);    /* +1 for null-terminator */

   string = (char*)memset(buffer, 'A', 10);
   printf("\nBuffer contents: %s\n", string);
   memset(buffer+10, 'B', 20);

   return 0;

   /*****************************************************************************
      The output should be:

      Buffer contents: AAAAAAAAAA
      End of allocated object 0x00073c80 was overwritten at 0x00073c95.
      The first eight bytes of the memory block (in hex) are: 4141414141414141.
      This memory block was (re)allocated at line number 12 in memset.c.
      Heap state was valid at line 14 of memset.c.
      Memory error detected at line 16 of memset.c.
   *****************************************************************************/
}
```

RELATED CONCEPTS

Debugging Memory Heaps (page 29)
Memory Management Functions (page 25)
Managing Memory with Multiple Memory Heaps (page 27)

RELATED TASKS

Debug Programs with Heap Memory (page 35)

RELATED REFERENCES

_debug_calloc - Allocate and Initialize Memory (page 265)
_debug_free - Free Allocated Memory (page 266)
_debug_heapmin - Free Unused Memory in the Default Heap (page 268)
_debug_malloc - Allocate Memory (page 269)
_debug_memcpy - Copy Bytes (page 271)
_debug_memmove - Copy Bytes (page 273)
_debug_realloc - Reallocate Memory Block (page 276)
_debug_strcat - Concatenate Strings (page 278)

_debug_strcpy - Copy Strings (page 279)
_debug_strncat - Concatenate Strings (page 282)
_debug_strncpy - Copy Strings (page 284)
_debug_strnset - Set Characters in String (page 281)
_debug_strset - Set Characters in String (page 285)
_debug_ucalloc - Reserve and Initialize Memory from User Heap (page 287)
_debug_uheapmin - Free Unused Memory in User Heap (page 289)
_debug_umalloc - Reserve Memory Block from User Heap (page 290)
-qheapdebug Compile Option

_debug_realloc — Reallocate Memory Block

Format

```
#include <stdlib.h>  /* also in <malloc.h> */
void *_debug_realloc(void *ptr, size_t size,
                     const char *file, size_t line);
```

Language Level: Extension
_debug_realloc is the debug version of realloc. Like realloc, it reallocates the block of memory pointed to by *ptr* to a new *size*, specified in bytes. It also sets any new memory it allocates to 0xAA, so you can easily locate instances where your program tries to use the data in that memory without initializing it first.

In addition, _debug_realloc makes an implicit call to _heap_check, and stores the file name *file* and the line number *line* where the storage is reallocated. This information can be used later by the _heap_check, _dump_allocated, or _dump_allocated_delta functions.

If *ptr* is NULL, _debug_realloc behaves like _debug_malloc (or malloc) and allocates the block of memory.

To use _debug_realloc, you must compile with the _qheapdebug option. This option maps all realloc calls to _debug_realloc.

Note: The -qheapdebug option maps all calls to memory management functions (including heap-specific versions) to their debug counterparts. To prevent a call from being mapped, parenthesize the function name.

Because _debug_realloc always checks what heap the memory was allocated from, you can use _debug_realloc to reallocate memory blocks allocated by the regular or debug versions of the memory management functions. However, if the memory was not allocated by the memory management functions, or was previously freed, _debug_realloc generates an error message and the program ends.

Return Value
_debug_realloc returns a pointer to the reallocated memory block. The *ptr* argument to _debug_realloc is not the same as the return value; _debug_realloc always changes the memory location to help you locate references to the memory that were not freed before the memory was reallocated.

If *size* is 0, _debug_realloc returns NULL. If not enough memory is available to expand the block to the given size, the original block is unchanged and NULL is returned.

Example
This example uses _debug_realloc to allocate 100 bytes of storage. It then attempts

to write to storage that was not allocated. When _debug_realloc is called again, _heap_check detects the error, generates several messages, and stops the program.

Note: You must compile this example with alloc(debug,yes) to map the realloc calls to _debug_realloc.

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

int main(void)
{
   char *ptr;

   if (NULL == (ptr = (char*)realloc(NULL, 100))) {
      puts("Could not allocate memory block.");
      exit(EXIT_FAILURE);
   }
   memset(ptr, 'a', 105);     /* overwrites storage that was not allocated    */
   ptr = (char*)realloc(ptr, 200);        /*  realloc invokes _heap_check     */
   puts("_debug_realloc did not detect that a memory block was overwritten." );
   return 0;

   /****************************************************************************
      The output should be similar to:

      End of allocated object 0x00073890 was overwritten at 0x000738f4.
      The first eight bytes of the memory block (in hex) are: 6161616161616161.
      This memory block was (re)allocated at line number 8 in _debug_reall.c.
      Heap state was valid at line 8 of _debug_reall.c.
      Memory error detected at line 13 of _debug_reall.c.
   ****************************************************************************/
}
```

RELATED CONCEPTS

Debugging Memory Heaps (page 29)
Memory Management Functions (page 25)
Managing Memory with Multiple Memory Heaps (page 27)

RELATED TASKS

Debug Programs with Heap Memory (page 35)

RELATED REFERENCES

_debug_calloc - Allocate and Initialize Memory (page 265)
_debug_free - Free Allocated Memory (page 266)
_debug_heapmin - Free Unused Memory in the Default Heap (page 268)
_debug_malloc - Allocate Memory (page 269)
_debug_memcpy - Copy Bytes (page 271)
_debug_memmove - Copy Bytes (page 273)
_debug_memset - Set Bytes to Value (page 274)
_debug_strcat - Concatenate Strings (page 278)
_debug_strcpy - Copy Strings (page 279)
_debug_strncat - Concatenate Strings (page 282)
_debug_strncpy - Copy Strings (page 284)
_debug_strnset - Set Characters in String (page 281)
_debug_strset - Set Characters in String (page 285)
_debug_ucalloc - Reserve and Initialize Memory from User Heap (page 287)
_debug_uheapmin - Free Unused Memory in User Heap (page 289)
_debug_umalloc - Reserve Memory Block from User Heap (page 290)
-qheapdebug Compile Option

_debug_strcat — Concatenate Strings

Format

```
#include <string.h>

char *_debug_strcat(char *string1, const char *string2,

                    const char *file, size_t file);
```

Language Level: Extension
_debug_strcat is the debug version of strcat. Like strcat, it concatenates *string2* to *string1* and ends the resulting string with the null character.

_debug_strcat validates the heap after concatenating the strings, and performs this check only when the target is within a heap. _debug_strcat makes an implicit call to _heap_check. If _debug_strcat detects a corrupted heap when it makes a call to _heap_check, _debug_strcat will report the file name *file* and line number *file* in a message.

Note: _debug_strcat checks only the current default heap. Therefore, this debug support will not check all heaps within applications that have multiple user heaps.

To use _debug_strcat, you must compile with the -qheapdebug option. This option maps all strcat calls to _debug_strcat. You do not have to change your source code, in order for _debug_strcat to verify the heap.

Note: -qheapdebug maps all calls to other string functions and all calls to memory management functions (including a heap-specific version), to their debug counterparts. To prevent a call from being mapped, parenthesize the function name.

Return Value
_debug_strcat returns a pointer to the concatenated string *string1*.

Example
This example contains a programming error. The buffer1 object is not large enough to store the result after the string “ program” is concatenated.

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

#define  SIZE    10

int main(void)
{
   char *buffer1;
   char *ptr;

   buffer1 = (char*)malloc(SIZE);
   strcpy(buffer1, "computer");

   ptr = strcat(buffer1, " program");
   printf("buffer1 = %s\n", buffer1);
   return 0;

   /*****************************************************************************

      The output should be similar to:

      End of allocated object 0x00073c80 was overwritten at 0x00073c8a.
      The first eight bytes of the memory block (in hex) are: 636F6D7075746572.
      This memory block was (re)allocated at line number 12 in strcat.c.
      Heap state was valid at line 13 of strcat.c.
```

```
     Memory error detected at line 15 of strcat.c.

   *************************************************************************/
}
```

RELATED CONCEPTS

Debugging Memory Heaps (page 29)
Memory Management Functions (page 25)
Managing Memory with Multiple Memory Heaps (page 27)

RELATED TASKS

Debug Programs with Heap Memory (page 35)

RELATED REFERENCES

_debug_calloc - Allocate and Initialize Memory (page 265)
_debug_free - Free Allocated Memory (page 266)
_debug_heapmin - Free Unused Memory in the Default Heap (page 268)
_debug_malloc - Allocate Memory (page 269)
_debug_memcpy - Copy Bytes (page 271)
_debug_memmove - Copy Bytes (page 273)
_debug_memset - Set Bytes to Value (page 274)
_debug_realloc - Reallocate Memory Block (page 276)
_debug_strcpy - Copy Strings (page 279)
_debug_strncat - Concatenate Strings (page 282)
_debug_strncpy - Copy Strings (page 284)
_debug_strnset - Set Characters in String (page 281)
_debug_strset - Set Characters in String (page 285)
_debug_ucalloc - Reserve and Initialize Memory from User Heap (page 287)
_debug_uheapmin - Free Unused Memory in User Heap (page 289)
_debug_umalloc - Reserve Memory Block from User Heap (page 290)
-qheapdebug Compile Option

_debug_strcpy — Copy Strings

Format

```
#include <string.h>
char *_debug_strcpy(char *string1, const char *string2,
                    const char *file, size_t line);
```

Language Level: Extension

_debug_strcpy is the debug version of strcpy. Like strcpy, it copies `string2`, including the ending null character, to the location specified by `string1`.

_debug_strcpy validates the heap after copying the string to the target location, and performs this check only when the target is within a heap. _debug_strcpy makes an implicit call to _heap_check. If _debug_strcpy detects a corrupted heap when it makes a call to _heap_check, _debug_strcpy will report the file name `file` and line number `line` in a message.

Note: _debug_strcpy checks only the current default heap. Therefore, this debug support will not check all heaps within applications that have multiple user heaps.

To use _debug_strcpy, you must compile with the -qheapdebug option. This option maps all strcpy calls to _debug_strcpy. You do not have to change your source code in order for _debug_strcpy to verify the heap.

Note: -qheapdebug maps all calls to other string functions and all calls to memory management functions (including a heap-specific version), to their debug counterparts. To prevent a call from being mapped, parenthesize the function name.

Return Value
_debug_strcpy returns a pointer to the copied string *string1*.

Example
This example contains a programming error. The source string is too long for the destination buffer, and the strcpy operation damages the heap.

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

#define  SIZE         10

int main(void)
{
   char *source = "1234567890123456789";
   char *destination;
   char *return_string;

   destination = (char*)malloc(SIZE);
   strcpy(destination, "abcdefg"),

   printf("destination is originally = '%s'\n", destination);
   return_string = strcpy(destination, source);
   printf("After strcpy, destination becomes '%s'\n\n", destination);
   return 0;

   /****************************************************************************
      The output should be similar to:

      destination is originally = 'abcdefg'
      End of allocated object 0x00073c80 was overwritten at 0x00073c8a.
      The first eight bytes of the memory block (in hex) are: 3132333435363738.
      This memory block was (re)allocated at line number 13 in strcpy.c.
      Heap state was valid at line 14 of strcpy.c.
      Memory error detected at line 17 of strcpy.c.
   ****************************************************************************/
}
```

RELATED CONCEPTS

Debugging Memory Heaps (page 29)
Memory Management Functions (page 25)
Managing Memory with Multiple Memory Heaps (page 27)

RELATED TASKS

Debug Programs with Heap Memory (page 35)

RELATED REFERENCES

_debug_calloc - Allocate and Initialize Memory (page 265)
_debug_free - Free Allocated Memory (page 266)
_debug_heapmin - Free Unused Memory in the Default Heap (page 268)
_debug_malloc - Allocate Memory (page 269)
_debug_memcpy - Copy Bytes (page 271)
_debug_memmove - Copy Bytes (page 273)
_debug_memset - Set Bytes to Value (page 274)
_debug_realloc - Reallocate Memory Block (page 276)
_debug_strcat - Concatenate Strings (page 278)
_debug_strncat - Concatenate Strings (page 282)
_debug_strncpy - Copy Strings (page 284)
_debug_strnset - Set Characters in String (page 281)

_debug_strset - Set Characters in String (page 285)
_debug_ucalloc - Reserve and Initialize Memory from User Heap (page 287)
_debug_uheapmin - Free Unused Memory in User Heap (page 289)
_debug_umalloc - Reserve Memory Block from User Heap (page 290)
-qheapdebug Compile Option

_debug_strnset — Set Characters in String

Format

```
#include <string.h>
char *_debug_strnset(char *string, int c, size_t n,
                     const char *file, size_t line);
```

Language Level: Extension
_debug_strnset is the debug version of strnset. Like strnset, it sets, at most, the first *n* characters of *string* to *c* (converted to a char), where if *n* is greater than the length of *string*, the length of *string* is used in place of *n*.

_debug_strnset validates the heap after setting the bytes, and performs this check only when the target is within a heap. _debug_strnset makes an implicit call to _heap_check. If _debug_strnset detects a corrupted heap when it makes a call to _heap_check, _debug_strnset will report the file name *file* and line number *line* in a message.

Note: _debug_strnset checks only the current default heap. Therefore, this debug support will not check all heaps within applications that have multiple user heaps.

To use _debug_strnset, you must compile with the debug memory -qheapdebug option. This option maps all strnset calls to _debug_strnset. You do not have to change your source code in order for _debug_memset to verify the heap.

Note: -qheapdebug maps all calls to memory management functions (including a heap-specific version) to their debug counterparts. To prevent a call from being mapped, parenthesize the function name.

Return Value
_debug_strnset returns a pointer to the altered *string*. There is no error return value.

Example
This example contains two programming errors. The string, str, was created without a null-terminator to mark the end of the string, and without the terminator strnset with a count of 10 stores bytes past the end of the allocated object.

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

int main(void)
{
   char *str;

   str = (char*)malloc(10);

   printf("This is the string after strnset: %s\n", str);
   return 0;

   /****************************************************************************
      The output should be:
```

```
      End of allocated object 0x00073c80 was overwritten at 0x00073c8a.
      The first eight bytes of the memory block (in hex) are: 7878787878797979.
      This memory block was (re)allocated at line number 9 in strnset.c.
      Heap state was valid at line 11 of strnset.c.
   **************************************************************************/
}
```

RELATED CONCEPTS

Debugging Memory Heaps (page 29)
Memory Management Functions (page 25)
Managing Memory with Multiple Memory Heaps (page 27)

RELATED TASKS

Debug Problems with Heap Memory (page 35)

RELATED REFERENCES

_debug_calloc - Allocate and Initialize Memory (page 265)
_debug_free - Free Allocated Memory (page 266)
_debug_heapmin - Free Unused Memory in the Default Heap (page 268)
_debug_malloc - Allocate Memory (page 269)
_debug_memcpy - Copy Bytes (page 271)
_debug_memmove - Copy Bytes (page 273)
_debug_memset - Set Bytes to Value (page 274)
_debug_realloc - Reallocate Memory Block (page 276)
_debug_strcat - Concatenate Strings (page 278)
_debug_strcpy - Copy Strings (page 279)
_debug_strncat - Concatenate Strings (page 282)
_debug_strncpy - Copy Strings (page 284)
_debug_strset - Set Characters in String (page 285)
_debug_ucalloc - Reserve and Initialize Memory from User Heap (page 287)
_debug_uheapmin - Free Unused Memory in User Heap (page 289)
_debug_umalloc - Reserve Memory Block from User Heap (page 290)
-qheapdebug Compile Option

_debug_strncat — Concatenate Strings

Format

```
#include <string.h>
char *_debug_strncat(char *string1, const char *string2, size_t count,
                     const char *file, size_t line);
```

Language Level: Extension

_debug_strncat is the debug version of strncat. Like strncat, it appends the first count characters of *string2* to *string1* and ends the resulting string with a null character (\0). If *count* is greater than the length of *string2*, the length of *string2* is used in place of *count*.

_debug_strncat validates the heap after appending the characters, and performs this check only when the target is within a heap. _debug_strncat makes an implicit call to _heap_check. If _debug_strncat detects a corrupted heap when it makes a call to _heap_check, _debug_strncat will report the file name *file* and line number *line* in a message.

Note: _debug_strncat checks only the current default heap. Therefore, this debug support will not check all heaps within applications that have multiple user heaps.

To use _debug_strncat, you must compile with the -qheapdebug option. This option maps all strncat calls to _debug_strncat. You do not have to change your source code, in order for _debug_strncat to verify the heap.

Note: -qheapdebug maps all calls to other string functions and all calls to memory management functions (including a heap-specific version), to their debug counterparts. To prevent a call from being mapped, parenthesize the function name.

Return Value
_debug_strncat returns a pointer to the joined string *string1*.

Example
This example contains a programming error. The `buffer1` object is not large enough to store the result after eight characters from the string `" programming"` are concatenated.

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

#define  SIZE         10

int main(void)
{
   char *buffer1;
   char *ptr;

   buffer1 = (char*)malloc(SIZE);
   strcpy(buffer1, "computer");

   /* Call strncat with buffer1 and " programming"                           */

   ptr = strncat(buffer1, " programming", 8);
   printf("strncat: buffer1 = \"%s\"\n", buffer1);
   return 0;

   /**************************************************************************
      The output should be similar to:

      End of allocated object 0x00073c80 was overwritten at 0x00073c8a.
      The first eight bytes of the memory block (in hex) are: 636F6D7075746572.
      This memory block was (re)allocated at line number 12 in strncat.c.
      Heap state was valid at line 13 of strncat.c.
      Memory error detected at line 17 of strncat.c.
   **************************************************************************/
}
```

RELATED CONCEPTS
Debugging Memory Heaps (page 29)
Memory Management Functions (page 25)
Managing Memory with Multiple Memory Heaps (page 27)

RELATED TASKS
Debug Programs with Heap Memory (page 35)

RELATED REFERENCES
_debug_calloc - Allocate and Initialize Memory (page 265)
_debug_free - Free Allocated Memory (page 266)
_debug_heapmin - Free Unused Memory in the Default Heap (page 268)
_debug_malloc - Allocate Memory (page 269)
_debug_memcpy - Copy Bytes (page 271)
_debug_memmove - Copy Bytes (page 273)
_debug_memset - Set Bytes to Value (page 274)
_debug_realloc - Reallocate Memory Block (page 276)

_debug_strcat - Concatenate Strings (page 278)
_debug_strcpy - Copy Strings (page 279)
_debug_strncpy - Copy Strings (page 284)
_debug_strnset - Set Characters in String (page 281)
_debug_strset - Set Characters in String (page 285)
_debug_ucalloc - Reserve and Initialize Memory from User Heap (page 287)
_debug_uheapmin - Free Unused Memory in User Heap (page 289)
_debug_umalloc - Reserve Memory Block from User Heap (page 290)
-qheapdebug Compile Option

_debug_strncpy — Copy Strings

Format

```
#include <string.h>
char *_debug_strncpy(char *string1, const char *string2, size_t count,
                     const char *file, size_t line);
```

Language Level: Extension
_debug_strncpy is the debug version of **strncpy**. Like **strncpy**, it copies *count* characters of *string2* to *string1*. If *count* is less than or equal to the length of *string2*, a null character (\0) is not appended to the copied string. If *count* is greater than the length of *string2*, the *string1* result is padded with null characters (\0) up to length *count*.

_debug_strncpy validates the heap after copying the strings to the target location, and performs this check only when the target is within a heap. **_debug_strncpy** makes an implicit call to **_heap_check**. If **_debug_strncpy** detects a corrupted heap when it makes a call to **_heap_check**, **_debug_strncpy** will report the file name *file* and line number *line* in a message.

Note: _debug_strncpy checks only the current default heap. Therefore, this debug support will not check all heaps within applications that have multiple user heaps.

To use **_debug_strncpy**, you must compile with the -qheapdebug option. This option maps all **strncpy** calls to **_debug_strncpy**. You do not have to change your source code, in order for **_debug_strncpy** to verify the heap.

Note: The -qheapdebug option maps all calls to other string functions and all calls to memory management functions (including a heap-specific version), to their debug counterparts. To prevent a call from being mapped, parenthesize the function name.

Return Value
_debug_strncpy returns a pointer to *string1*.

Example
This example contains a programming error. The source string is too long for the destination buffer, and the **strncpy** operation damages the heap.

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

#define  SIZE         10
```

```
int main(void)
{
   char *source = "1234567890123456789";
   char *destination;
   char *return_string;
   int index = 15;

   destination = (char*)malloc(SIZE);
   strcpy(destination, "abcdefg"),

   printf("destination is originally = '%s'\n", destination);
   return_string = strncpy(destination, source, index);
   printf("After strncpy, destination becomes '%s'\n\n", destination);
   return 0;

   /****************************************************************************
      The output should be similar to:

      destination is originally = 'abcdefg'
      End of allocated object 0x00073c80 was overwritten at 0x00073c8a.
      The first eight bytes of the memory block (in hex) are: 3132333435363738.
      This memory block was (re)allocated at line number 14 in strncpy.c.
      Heap state was valid at line 15 of strncpy.c.
      Memory error detected at line 18 of strncpy.c.
   ****************************************************************************/
}
```

RELATED CONCEPTS

Debugging Memory Heaps (page 29)
Memory Management Functions (page 25)
Managing Memory with Multiple Memory Heaps (page 27)

RELATED TASKS

Debug Programs with Heap Memory (page 35)

RELATED REFERENCES

_debug_calloc - Allocate and Initialize Memory (page 265)
_debug_free - Free Allocated Memory (page 266)
_debug_heapmin - Free Unused Memory in the Default Heap (page 268)
_debug_malloc - Allocate Memory (page 269)
_debug_memcpy - Copy Bytes (page 271)
_debug_memmove - Copy Bytes (page 273)
_debug_memset - Set Bytes to Value (page 274)
_debug_realloc - Reallocate Memory Block (page 276)
_debug_strcat - Concatenate Strings (page 278)
_debug_strcpy - Copy Strings (page 279)
_debug_strncat - Concatenate Strings (page 282)
_debug_strnset - Set Characters in String (page 281)
_debug_strset - Set Characters in String (page 285)
_debug_ucalloc - Reserve and Initialize Memory from User Heap (page 287)
_debug_uheapmin - Free Unused Memory in User Heap (page 289)
_debug_umalloc - Reserve Memory Block from User Heap (page 290)
-qheapdebug Compile Option

_debug_strset — Set Characters in String

Format

```
#include <string.h>
char *_debug_strset(char *string, size_t c,
                    const char *file, size_t line);
```

Language Level: Extension
_debug_strset is the debug version of strset. Like strset, it sets all characters of *string*, except the ending null character (\0), to *c* (converted to a char).

_debug_strset validates the heap after setting all characters of *string*, and performs this check only when the target is within a heap. **_debug_strset** makes an implicit call to **_heap_check**. If **_debug_strset** detects a corrupted heap when it makes a call to **_heap_check**, **_debug_strset** will report the file name *file* and line number *line* in a message.

Note: **_debug_strset** checks only the current default heap. Therefore, this debug support will not check all heaps within applications that have multiple user heaps.

To use **_debug_strset**, you must compile with the debug memory -qheapdebug option. This option maps all **strset** calls to **_debug_strset**. You do not have to change your source code, in order for **_debug_strset** to verify the heap.

Note: -qheapdebug maps all calls to other string functions and all calls to memory management functions (including a heap-specific version), to their debug counterparts. To prevent a call from being mapped, parenthesize the function name.

Return Value
_debug_strset returns a pointer to the altered *string*. There is no error return value.

Example
This example contains a programming error. The string, `str`, was created without a null-terminator, and **strset** propagates the letter 'k' until it finds what it thinks is the null-terminator.

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

int main(void)
{
   char *str;

   str = (char*)malloc(10);

   strnset(str, 'x', 5);
   strset(str+5, 'k');
   printf("This is the string after strset: %s\n", str);
   return 0;

   /****************************************************************************
      The output should be:

      End of allocated object 0x00073c80 was overwritten at 0x00073c8a.
      The first eight bytes of the memory block (in hex) are: 78787878786B6B6B.
      This memory block was (re)allocated at line number 9 in strset.c.
      Heap state was valid at line 11 of strset.c.
      Memory error detected at line 12 of strnset.c.
   ****************************************************************************/
}
```

RELATED CONCEPTS
Debugging Memory Heaps (page 29)
Memory Management Functions (page 25)
Managing Memory with Multiple Memory Heaps (page 27)

RELATED TASKS

Debug Programs with Heap Memory (page 35)

RELATED REFERENCES

_debug_calloc - Allocate and Initialize Memory (page 265)
_debug_free - Free Allocated Memory (page 266)
_debug_heapmin - Free Unused Memory in the Default Heap (page 268)
_debug_malloc - Allocate Memory (page 269)
_debug_memcpy - Copy Bytes (page 271)
_debug_memmove - Copy Bytes (page 273)
_debug_memset - Set Bytes to Value (page 274)
_debug_realloc - Reallocate Memory Block (page 276)
_debug_strcat - Concatenate Strings (page 278)
_debug_strcpy - Copy Strings (page 279)
_debug_strncat - Concatenate Strings (page 282)
_debug_strncpy - Copy Strings (page 284)
_debug_strnset - Set Characters in String (page 281)
_debug_ucalloc - Reserve and Initialize Memory from User Heap (page 287)
_debug_uheapmin - Free Unused Memory in User Heap (page 289)
_debug_umalloc - Reserve Memory Block from User Heap (page 290)
-qheapdebug Compile Option

_debug_ucalloc — Reserve and Initialize Memory from User Heap

Format

```
#include <umalloc.h>
void *_debug_ucalloc(Heap_t heap, size_t num, size_t size,
                     const char *file, size_t line);
```

Language Level: Extension
_debug_ucalloc is the debug version of _ucalloc. Like _ucalloc, it allocates memory from the `heap` you specify for an array of `num` elements, each of length `size` bytes. It then initializes all bits of each element to 0.

In addition, _debug_ucalloc makes an implicit call to _uheap_check, and stores the name of the file `file` and the line number `line` where the storage is allocated. This information can be used later by the _uheap_check, _uheap_allocated, or _udump_allocated_delta functions.

To use _debug_ucalloc, you must compile with the -qheapdebug option. This option maps all _ucalloc calls to _debug_ucalloc.

Note: The -qheapdebug option maps all calls to memory management functions (including heap-specific versions) to their debug counterparts. To prevent a call from being mapped, parenthesize the function name.

_debug_ucalloc works just like _debug_calloc except that you specify the heap to use; _debug_calloc always allocates from the default heap.

If the `heap` does not have enough memory for the request, _debug_ucalloc calls the *getmore_fn* that you specified when you created the heap with _ucreate.

To reallocate or free memory allocated with _debug_ucalloc, use the non-heap-specific _debug_realloc and _debug_free. These functions always check what heap the memory was allocated from.

Return Value
_debug_ucalloc returns a pointer to the reserved space. If `size` or `num` was specified as zero, or if your *getmore_fn* cannot provide enough memory, _debug_ucalloc returns NULL. Passing _debug_ucalloc a heap that is not valid results in undefined behavior.

Example
This example creates a user heap and allocates memory from it with _debug_ucalloc. It then attempts to write to memory that was not allocated. When _debug_free is called, _uheap_check detects the error, generates several messages, and stops the program.

Note: You must compile this example with the -qheapdebug option to map the _ucalloc calls to _debug_ucalloc and free to _debug_free.

```
#include <stdlib.h>
#include <stdio.h>
#include <umalloc.h>
#include <string.h>

int main(void)
{
   Heap_t  myheap;
   char    *ptr;

   /* Use default heap as user heap */
   myheap = _udefault(NULL);

   if (NULL == (ptr = (char*)_ucalloc(myheap, 100, 1))) {
      puts("Cannot allocate memory from user heap.");
      exit(EXIT_FAILURE);
   }
   memset(ptr, 'x', 105);   /* Overwrites storage that was not allocated */
   free(ptr);
   return 0;

   /****************************************************************************
      The output should be similar to :

      End of allocated object 0x00073890 was overwritten at 0x000738f4.
      The first eight bytes of the memory block (in hex) are: 7878787878787878.
      This memory block was (re)allocated at line number 14 in _debug_ucallo.c.
      Heap state was valid at line 14 of _debug_ucallo.c.
      Memory error detected at line 19 of _debug_ucallo.c.
   ****************************************************************************/
}
```

RELATED CONCEPTS

Debugging Memory Heaps (page 29)
Memory Management Functions (page 25)
Managing Memory with Multiple Memory Heaps (page 27)

RELATED TASKS

Debug Programs with Heap Memory (page 35)

RELATED REFERENCES

_debug_calloc - Allocate and Initialize Memory (page 265)
_debug_free - Free Allocated Memory (page 266)
_debug_heapmin - Free Unused Memory in the Default Heap (page 268)
_debug_malloc - Allocate Memory (page 269)
_debug_memcpy - Copy Bytes (page 271)
_debug_memmove - Copy Bytes (page 273)
_debug_memset - Set Bytes to Value (page 274)
_debug_realloc - Reallocate Memory Block (page 276)

_debug_strcat - Concatenate Strings (page 278)
_debug_strcpy - Copy Strings (page 279)
_debug_strncat - Concatenate Strings (page 282)
_debug_strncpy - Copy Strings (page 284)
_debug_strnset - Set Characters in String (page 281)
_debug_strset - Set Characters in String (page 285)
_debug_uheapmin - Free Unused Memory in User Heap (page 289)
_debug_umalloc - Reserve Memory Block from User Heap (page 290)
-qheapdebug Compile Option

_debug_uheapmin — Free Unused Memory in User Heap

Format

```
#include <umalloc.h>
int _debug_uheapmin(Heap_t heap, const char *file, size_t line);
```

Language Level: Extension
_debug_uheapmin is the debug version of _uheapmin. Like _uheapmin, it returns all unused memory blocks from the specified *heap* to the operating system.

To return the memory, _debug_uheapmin calls the *release_fn* you supplied when you created the heap with _ucreate. If you do not supply a *release_fn*, _debug_uheapmin has no effect and returns 0.

In addition, _debug_uheapmin makes an implicit call to _uheap_check to validate the heap.

_debug_uheapmin works just like _debug_heapmin except that you specify the heap to use; _debug_heapmin always uses the default heap.

To use _debug_uheapmin, you must compile with the -qheapdebug option. This option maps all _uheapmin calls to _debug_uheapmin.

Note: -qheapdebug maps all calls to memory management functions (including heap-specific versions) to their debug counterparts. To prevent a call from being mapped, parenthesize the function name.

Return Value
If successful, _debug_uheapmin returns 0. A nonzero return value indicates failure. If the heap specified is not valid, _debug_uheapmin generates an error message with the file name and line number where the call to _debug_uheapmin was made.

Example
This example creates a heap and allocates memory from it, then uses _debug_heapmin to release the memory.

Note: You must compile this example with -qheapdebug to map the _uheapmin calls to _debug_uheapmin.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <umalloc.h>

int main(void)
{
   Heap_t  myheap;
   char    *ptr;
```

```
   /* Use default heap as user heap */
   myheap = _udefault(NULL);

   /* Allocate a large object */
   if (NULL == (ptr = (char*)_umalloc(myheap, 60000))) {
      puts("Cannot allocate memory from user heap.\n");
      exit(EXIT_FAILURE);
   }
   memset(ptr, 'x', 60000);
   free(ptr);

   /* _debug_uheapmin will attempt to return the freed object to the system */
   if (0 != _uheapmin(myheap)) {
      puts("_debug_uheapmin returns failed.\n");
      exit(EXIT_FAILURE);
   }
   return 0;
}
```

RELATED CONCEPTS

Debugging Memory Heaps (page 29)
Memory Management Functions (page 25)
Managing Memory with Multiple Memory Heaps (page 27)

RELATED TASKS

Debug Programs with Heap Memory (page 35)

RELATED REFERENCES

_debug_calloc - Allocate and Initialize Memory (page 265)
_debug_free - Free Allocated Memory (page 266)
_debug_heapmin - Free Unused Memory in the Default Heap (page 268)
_debug_malloc - Allocate Memory (page 269)
_debug_memcpy - Copy Bytes (page 271)
_debug_memmove - Copy Bytes (page 273)
_debug_memset - Set Bytes to Value (page 274)
_debug_realloc - Reallocate Memory Block (page 276)
_debug_strcat - Concatenate Strings (page 278)
_debug_strcpy - Copy Strings (page 279)
_debug_strncat - Concatenate Strings (page 282)
_debug_strncpy - Copy Strings (page 284)
_debug_strnset - Set Characters in String (page 281)
_debug_strset - Set Characters in String (page 285)
_debug_ucalloc - Reserve and Initialize Memory from User Heap (page 287)
_debug_umalloc - Reserve Memory Block from User Heap (page 290)
-qheapdebug Compile Option

_debug_umalloc — Reserve Memory Blocks from User Heap

Format

```
#include <umalloc.h>
void *_debug_umalloc(Heap_t heap, size_t size,
                     const char *file, size_t line);
```

Language Level: Extension

_debug_umalloc is the debug version of _umalloc. Like _umalloc, it reserves storage space from the *heap* you specify for a block of *size* bytes. _debug_umalloc also sets all the memory it allocates to 0xAA, so you can easily locate instances where your program uses the data in the memory without initializing it first.

In addition, _debug_umalloc makes an implicit call to _uheap_check, and stores the name of the file `file` and the line number `line` where the storage is allocated. This information can be used later by the _uheap_check, _udump_allocated, or _udump_allocated_delta functions. _debug_umalloc also sets all the memory it allocates to 0xAA; this can help you debug problems where your program uses the data in the memory without initializing it.

_debug_umalloc works just like _debug_malloc except that you specify the heap to use; _debug_malloc always allocates from the default heap.

If the *heap* does not have enough memory for the request, _debug_umalloc calls the *getmore_fn* that you specified when you created the heap with _ucreate.

To use _debug_umalloc, you must compile with the -qheapdebug option. This option maps all _umalloc calls to _debug_umalloc.

Note: -qheapdebug maps all calls to memory management functions (including heap-specific versions) to their debug counterparts. To prevent a call from being mapped, parenthesize the function name.

To reallocate or free memory allocated with _debug_umalloc, use the non-heap-specific _debug_realloc and _debug_free. These functions always check what heap the memory was allocated from.

Return Value

_debug_umalloc returns a pointer to the reserved space. If *size* was specified as zero, or the *getmore_fn* cannot provide enough memory, _debug_umalloc returns NULL. Passing _debug_umalloc a heap that is not valid results in undefined behavior.

Example

This example creates a heap myheap and uses _debug_umalloc to allocate 100 bytes from it. It then attempts to overwrite storage that was not allocated. The call to _debug_free invokes _uheap_check, which detects the error, generates messages, and ends the program.

Note: You must compile this example with the -qheapdebug to map _umalloc to _debug_umalloc, and free to _debug_free.

```
#include <stdlib.h>
#include <stdio.h>
#include <umalloc.h>
#include <string.h>

int main(void)
{
   Heap_t  myheap;
   char    *ptr;

   /* Use default heap as user heap */
   myheap = _udefault(NULL);

   if (NULL == (ptr = (char*)_umalloc(myheap, 100))) {
      puts("Cannot allocate memory from user heap.\n");
      exit(EXIT_FAILURE);
   }
   memset(ptr, 'x', 105);   /* Overwrites storage that was not allocated */
   free(ptr);
   return 0;

   /*****************************************************************************
      The output should be similar to :
```

```
      End of allocated object 0x00073890 was overwritten at 0x000738f4.
      The first eight bytes of the memory block (in hex) are: 7878787878787878.
      This memory block was (re)allocated at line number 14 in _debug_umallo.c.
      Heap state was valid at line 14 of _debug_umallo.c.
      Memory error detected at line 19 of _debug_umallo.c.
   *****************************************************************************/
}
```

RELATED CONCEPTS

Debugging Memory Heaps (page 29)
Memory Management Functions (page 25)
Managing Memory with Multiple Memory Heaps (page 27)

RELATED TASKS

Debug Programs with Heap Memory (page 35)

RELATED REFERENCES

_debug_calloc - Allocate and Initialize Memory (page 265)
_debug_free - Free Allocated Memory (page 266)
_debug_heapmin - Free Unused Memory in the Default Heap (page 268)
_debug_malloc - Allocate Memory (page 269)
_debug_memcpy - Copy Bytes (page 271)
_debug_memmove - Copy Bytes (page 273)
_debug_memset - Set Bytes to Value (page 274)
_debug_realloc - Reallocate Memory Block (page 276)
_debug_strcat - Concatenate Strings (page 278)
_debug_strcpy - Copy Strings (page 279)
_debug_strncat - Concatenate Strings (page 282)
_debug_strncpy - Copy Strings (page 284)
_debug_strnset - Set Characters in String (page 281)
_debug_strset - Set Characters in String (page 285)
_debug_ucalloc - Reserve and Initialize Memory from User Heap (page 287)
_debug_uheapmin - Free Unused Memory in User Heap (page 289)
-qheapdebug Compile Option

Contacting IBM

We're delighted to offer a solid cornerstone for your application development: IBM's comprehensive support services. Whether you are an occasional user with a simple question, a power user with many complex technical questions, or someone who requires application design assistance and consulting, IBM Support can meet your needs.

Comments on This Help

Please let us know about any errors or omissions in this online help or in the hardcopy Memo to Users, or our PDF documents. Send your e-mail to: compinfo@ca.ibm.com

Fee Support

Developers on the VisualAge C++ for AIX Service and Support team handle everything from simple how-to's to complex technical problems. Solutions may take the form of a brief explanation, a workaround, a fix to the current product, or a fix to the next release.

http://www.ibm.com/software/ad/vacpp/support.html describes IBM Support Offerings for VisualAge C++.

http://www.ibm.com/support/ describes IBM Support Offerings on all platforms, worldwide.

http://www.ibm.com/rs6000/support/ describes support offerings on the RS/6000 platform, in your country. It also indicates whether your country provides support electronically over the Internet in addition to telephone support.

http://www.lotus.com/passport describes the IBM and Lotus Passport Advantage contracting option.

The IBM Software Support Handbook, accessible from http://www.ibm.com/software/support, also lists worldwide support contacts.

Phone numbers for information on Support Line offerings are:

- United States: 1-888-426-4343 (IBM Global Services), option 3 as of December 1999. Should this number change, IBM general information at 1-800-IBM-4YOU (1-800-426-4968) can route you to the appropriate group.
- Canada: 1-800-465-9600, option 3 as of December 1999. Should this number change, you can also contact IBM general information at 1-800-IBM-4YOU (1-800-426-4968).

Please call 1-800-266-8720 in the U.S. and Canada for information on Passport Advantage offerings.

Elsewhere, please contact your local IBM office.

© Copyright IBM Corp. 1998, 2002

If you contact support, please have the following information available:

- The product name
- The product version
- The hardware configuration and software (product names and versions you are using)
- What happened and what you were doing when the problem occurred
- Whether you tried to solve the problem and how
- The exact wording of any messages displayed

Consulting Services

VisualAge and WebSphere Product Affinity Services Group is a core group of technical specialists from the IBM development labs that created the IBM VisualAge and WebSphere products. With access to a network of IBM product area experts, IBM and industry business partners, and some of the best resources in the industry, we can put the optimal team in place to meet the challenge of absorbing new technology. Our goal is to enable organizational success with VisualAge and WebSphere — ensuring that our products are used effectively within your development team.

For more information, visit http://www.ibm.com/software/ad/vaws-services/ or contact the Product Affinity Services Team at:

AIM_SERVICES@us.ibm.com