C/C++ Legacy Class Libraries Reference

SC09-7652-02

C/C++ Legacy Class Libraries Reference

SC09-7652-02

Note!
Before using this information and the product it supports, read the information in "Notices" on page 203.

Second Edition (September 2004)

IBM welcomes your comments. You can send them to compinfo@ca.ibm.com. Be sure to include your e-mail address if you want a reply. Include the title and order number of this book, and the page number or topic related to your comment.

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

Contents

Preface

Previous releases of the IBM C/C++ compiler on z/OS, AIX, and OS/400 included support for the IBM Open Class (IOC) Library. This support has been removed from the compiler or will be removed from the compiler.

The UNIX System Laboratories (USL) I/O Stream Library and Complex Mathematics Library are still supported on z/OS, AIX, and OS/400. Although support for these classes is not being removed at this time, it is recommended that you migrate to the Standard C++ iostream and complex classes. This is especially important if you are migrating other IOC streaming classes to Standard C++ Library streaming classes, because combining USL and Standard C++ Library streams in one application is not recommended.

This manual provides information about the USL I/O Stream Library and the Complex Mathematics Library. For information about how to migrate away from these classes, see the *IBM Open Class Library Transition Guide*.

The following symbols indicate information that is specific to AIX, OS/400, or z/OS: 

Chapter 1. USL I/O Streaming

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

We recommend that you use the standard C++ stream classes instead of the USL I/O Stream Library to develop thread-safe applications. For more information about the Standard C++ I/O Stream Library, see the *Standard C++ Library Reference*.

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.

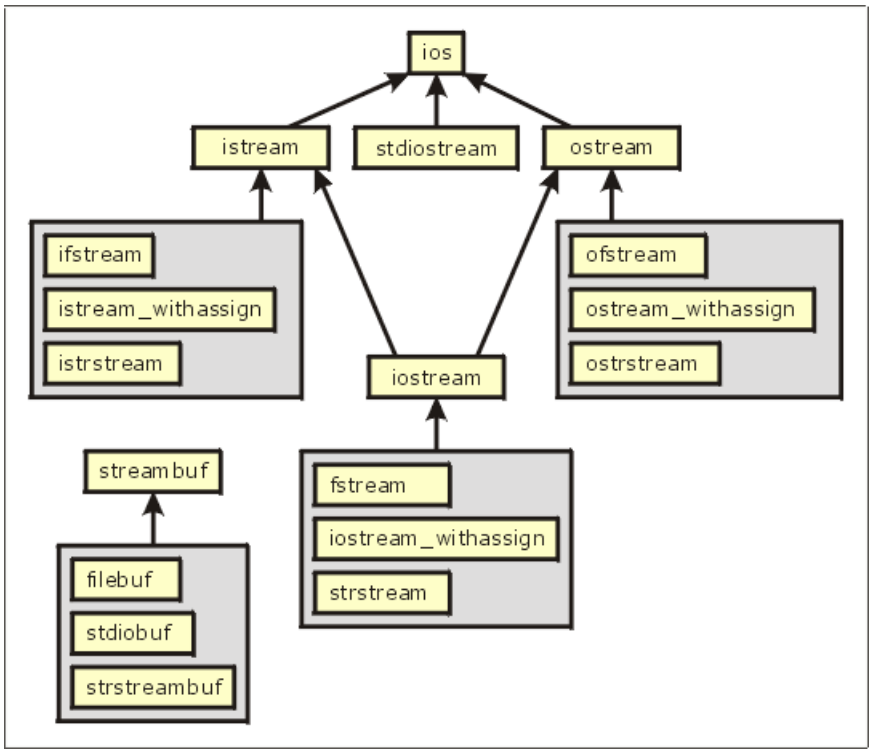
The USL I/O Stream Library predefines streams for standard input, standard output, and standard error. If you want to open your own streams for input or output, you must create an object of an appropriate I/O Stream class. The iostream constructor takes as an argument a pointer to a streambuf object. This object is associated with the device, file, or array of bytes in memory that is going to be the ultimate producer of input or the ultimate consumer of output.

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.

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:

- filebuf
- stdiobuf
- strstreambuf

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:

- istream
- stdiostream
- 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:

- ifstream
- istream_withassign
- istrstream
- iostream

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

- ofstream

- ostream_withassign
- ostrstream
- iostream

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

- fstream
- iostream_withassign
- strstream

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.

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.

The 8.3 file naming convention compliant name of this file is stdiostr.h. Under IFS, you can use either the short name or the long name (stdiostream.h).

The header file strstream.h contains declarations for the classes that deal with character strings.

The 8.3 file naming convention compliant name of this file is strstrea.h. Under IFS, you can use either the short name or the long name (strstream.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.

The USL I/O Stream Classes and stdio.h

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

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

Use Predefined Streams

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

- cin is the standard input stream.

  AIX z/OS file descriptor = 0.
- cout is the standard output stream.

  AIX z/OS file descriptor = 1.
- cerr is the standard error stream. Output to this stream is *unit-buffered*. Characters sent to this stream are flushed after each output operation.

  AIX z/OS file descriptor = 2.

- clog is also an error stream, but unlike the output to cerr, the output to clog is stream-buffered. Characters sent to this stream are flushed only when the stream becomes full or when it is explicitly flushed.

  AIX z/OS file descriptor = 2.

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

The predefined streams cin, cerr, and clog are *tied* to cout. As a result, if you use cin, cerr, or clog, cout is *flushed.* That is, the contents of cout are sent to their ultimate consumer.

Use Anonymous Streams

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

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

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

```
// Using anonymous streams

#include <fstream.h>

class MyClass {
public:
  char a;
};

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

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

// 1. Use an anonymous stream with a built-in type; this works

void f() {
  char a = 'a';

  // write to the file
  fstream("file1.abc",ios::out) << a << endl;
```

```
  // read from the file
  fstream("file1.abc",ios::in) >> a;

  // show what was in the file
  cout << a << endl;
}

#ifdef anon

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

void g() {
  MyClass b;
  b.a ='b';

  // write to the file
  fstream("file1.abc",ios::out) << b << endl;

  // read from the file
  fstream("file1.abc",ios::in) >> b;

  // show what was in the file
  cout << b << endl;
}

#endif

// 3. Use a named stream with a class type; this works

void h() {
  MyClass c;
  c.a ='c';

  // define and open the file
  fstream File2("file2.abc",ios::out);

  // write to the file
  File2 << c << endl;

  //close the file
  File2.close();

  // reopen for input
  File2.open("file2.abc",ios::in);

  // read from the file
  File2 >> c;

  // show what was in the file
  cout << c << endl;
}

int main(int argc, char *argv[]) {
  f();
#ifdef anon
  g();
#endif
  h();
  return 0;
}
```

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

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

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

Stream Buffers

One of the most important concepts in the 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 input from the user, a file, a device, or an array of bytes in memory. The ultimate consumer can 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, performance may improve. 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 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.

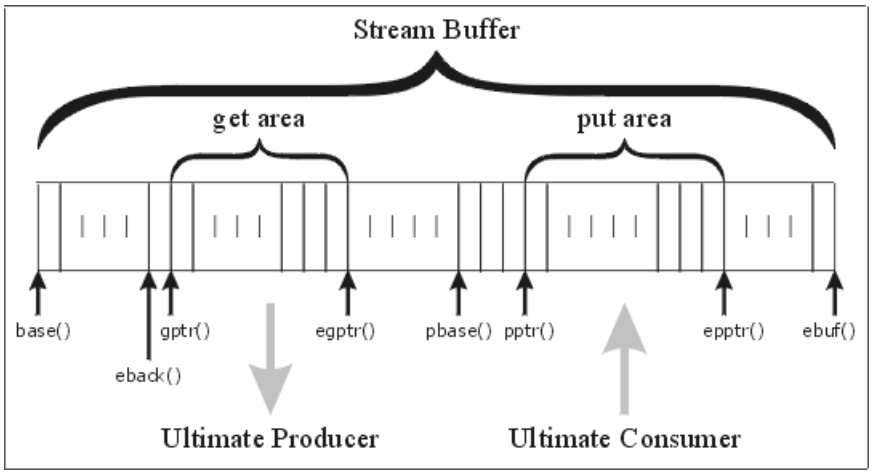
- A stream buffer that is used for both input and output, such as the one that is attached to an iostream object, has both a get area and a put area.
- In stream buffers implemented by the filebuf class that are specialized to use files as an ultimate producer or ultimate consumer, the get and put areas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

The following member functions of the streambuf class return pointers to get and put 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:



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

Format Stream Output

The USL I/O Stream Library lets you define how output should be formatted on a stream-by-stream basis within your program. Most formatting applies to numeric data: what base integers should be written to the output stream in, how many digits of precision floating-point numbers should have, whether they should appear in scientific or fixed-point format. Other formatting applies to any of the built-in types, and to your own types if you design your class output operators to check the format state of a stream to determine what formatting action to take.

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

ios Methods and Manipulators

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

```
#include <iostream.h>
int main(int argc, char *argv[]) {
   int a=9;
   cout.setf(ios::oct,ios::basefield);
   cout << a << endl;
// assume format state gets changed here, so we must change it back
   cout << oct << a << endl;
   return 0;
   }
```

Note that you do not need to qualify a manipulator, provided you do not create a variable or function of the same name as the manipulator. If a variable oct were declared at the start of the above example, cout << oct ... would write the variable oct to standard output. cout << ios::oct ... would change the format state.

Use setf, unsetf, and flags

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

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

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

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

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

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

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

#include <iostream.h>

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

int main(int argc, char *argv[]) {
// get and display current flag settings using flags()
   cout << "flags(): ";
   long originalFlags = cout.flags();
   showFlags();
```

```
// change format state using flags(long)
   cout << "flags(ios::fixed): ";
   cout.flags(ios::fixed);
   showFlags();
// change adjust field using setf(long)
   cout << "setf(ios::right): ";
   cout.setf(ios::right);
   showFlags();
// change floatfield using setf(long, long)
   cout << "setf(ios::scientific | ios::left,\n"
        << "ios::floatfield | ios::adjustfield): ";
   cout.setf(ios::scientific | ios::left,ios::floatfield |ios::adjustfield);
   showFlags();
// reset to original setting
   cout << "flags(originalFlags): ";
   cout.flags(originalFlags);
   showFlags();
   return 0;
 }
```

This example produces the following output:

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

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

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

Change the Notation of Floating-Point Values

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

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

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

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

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

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

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

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

```
// How format state flags and precision() affect output
#include <iostream.h>
int main(int argc, char *argv[]) {
   double a=3.14159265358979323846;
   double b;
   long originalFlags=cout.flags();
   int originalPrecision=cout.precision();
   for (double exp=1.;exp<1.0E+25;exp*=100000000.) {
      cout << "Printing new value for b:\n";
      b=a*exp; // Initialize b to a larger magnitude of a
// Now print b in a number of ways:
   // In fixed decimal notation
   cout.setf(ios::fixed,ios::floatfield);
   cout << " " << b << '\n';
   // In scientific notation
   cout.setf(ios::scientific,ios::floatfield);
   cout << " " <<b << '\n';
   // Change the exponent from lower to uppercase
   cout.setf(ios::uppercase);
   cout << " " <<b << '\n';
   // With 12 digits of precision, scientific notation
   cout.precision(12);
   cout << " " <<b << '\n';
   // Same precision, fixed notation
   cout.setf(ios::fixed,ios::floatfield);
   // Now set everything back to defaults
   cout.flags(originalFlags);
   cout.precision(originalPrecision);
   }
   return 0;
}
```

The output from this program is:

```
Printing new value for b:
   3.141593
   3.141593e+00
   3.141593E+00
   3.141592653590E+00
Printing new value for b:
   314159265.358979
   3.141593e+08
   3.141593E+08
   3.141592653590E+08
Printing new value for b:
   31415926535897932.000000
   3.141593e+16
   3.141593E+16
   3.141592653590E+16
Printing new value for b:
   3141592653589792849657856.000000
   3.141593e+24
   3.141593E+24
   3.141592653590E+24
```

Change the Base of Integral Values

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

```
//Showing the base of integer values

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

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

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

Set the Width and Justification of Output Fields

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

```
#include <iostream.h>
#include <iomanip.h>    // required for use of setw()
int main(int argc, char *argv[]) {
    int i=-5,j=7,k=-9;
    cout << setw(5) << i << setw(5) << j << setw(5) << k << endl;
    return 0;
}
```

You can also specify how values should be formatted within their fields. If the current width setting is greater than the number of characters required for the output, you can choose between right justification (the default), left justification, or,

for numeric values, internal justification (the sign, if any, is left-justified, while the value is right-justified). For example, the output statement above could be replaced with:

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

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

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

Define Your Own Format State Flags

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

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

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

```
// Defining your own format flags

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

long InFileFormat=0;
long OutFileFormat=0;

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

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

```
      << cust.City << '\n';
   if (os.flags() & OutFileFormat)    // New file format
      os << cust.Country << '\n'
         << cust.PostCode << endl;
   else                               // Old file format
      os << cust.PostCode << '\n'
         << cust.Country << endl;
   return os;
   }
istream& operator>>(istream &is, CustRecord &cust) {
   is >> cust.Number;
   is.ignore(1000,'\n'); // Ignore anything up to and including new line
   is.getline(cust.Name,48);
   is.getline(cust.Phone,16);
   is.getline(cust.Street,128);
   is.getline(cust.City,64);
   if (is.flags() & InFileFormat) {   // New file format!
      is.getline(cust.Country,64);
      is.getline(cust.PostCode,10);
      }
   else {
      is.getline(cust.PostCode,10);
      is.getline(cust.Country,64);
      }
   return is;
   }
int main(int argc, char* argv[]) {
   if (argc!=3) {                        // Requires two parameters
      cerr << "Specify an input file and an output file\n";
      exit(1);
      }
   ifstream InFile(argv[1]);
   ofstream OutFile(argv[2],ios::out);
   InFileFormat = InFile.bitalloc();     // Allocate flags for
   OutFileFormat = OutFile.bitalloc();   // each fstream
   if (InFileFormat==0 ||                // Exit if no flag could
       OutFileFormat==0) {               // be allocated
      cerr << "Could not allocate a user-defined format flag.\n";
      exit(2);
      }
   if (InFile.peek()=='!') {             // '!' means new format
      InFile.setf(InFileFormat);         // Input file is in new format
      OutFile.unsetf(OutFileFormat);     // Output file is in old format
      InFile.get();                      // Clear that first byte
      }
   else {                                // Otherwise, write '!' to
      OutFile << '!';                    // the output file, set the
      OutFile.setf(OutFileFormat);       // output stream's flag, and
      InFile.unsetf(InFileFormat);       // clear the input stream's
      }                                  // flag
   CustRecord record;
   while (InFile.peek()!=EOF) {         // Now read the input file
      InFile >> record;                 // records and write them
      OutFile << record;                // to the output file,
      }
   InFile.close();                      // Close both files
   OutFile.close();
   return 0;
 }
```

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

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

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

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.

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.

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
```

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
```

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;
}
```

Thread Safety and USL I/O Streaming

z/OS 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.

AIX 400 It is not safe to access a stream object in one thread while modifying it in another thread.

In a multi-threaded environment, there is no guarantee that the output from one streaming operator on the same thread will appear immediately after the output from the preceding streaming operator. For example, given the following scenario, either result may occur:

Scenario:

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

Result:

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

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

Basic USL I/O Stream Tasks

Receive Input from Standard Input

When you specify the iostream.h header file as a source file for your project, four streams are automatically defined for I/O use: cin, cout, cerr, and clog. The cin stream is the standard input stream. Input to cin comes from the C standard input stream, stdin, unless cin has been redirected by the user. The remaining streams can be used for output. You can receive standard input using the predefined input stream and the input operator (operator>>) for the type being read. In the following example, an integer is read from the input stream into a variable:

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

An input operator must exist for the type being read in. The USL I/O Stream Library defines input operators for all C++ built-in types. For types you define yourself, you need to provide your own input operators. If you attempt to read input into a variable and no input operator is defined for the type of that variable, the compiler displays an error message with text similar to the following:

```
Call does not match any parameter list for "operator>>".
```

Use Input Streams other than cin

You can use the same techniques for input from other input streams as for input from cin. The only difference is that, for other input streams, your program must define the stream. Suppose that you have defined a stream attached to a file opened for input, and have called that stream `myin`. You can read into `myin` from the file by specifying `myin` instead of cin:

```
// assume that the input file is associated
// with stream myin


int a, b;
myin >> a >> b;
```

Multiple Variables in an Input Statement

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

```
int i, j, k;
float m, n;

cin >> i >> j >> k >> m >> n;
```

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

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

```
cin >> j;
cin >> k;
cin >> m;
cin >> n;
```

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

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

String Input

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

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

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

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

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

White Space in String Input

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

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

#include <iostream.h>

int main(int argc, char *argv[]) {
   char text1[100], text2[100];

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

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

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

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

```
// get a line of text into array text2
cin.getline(text2, sizeof(text2), '\n');
// display the text arrays
cout << "<" << text1 << ">\n"
     << "<" << text2 << ">" << endl;
return 0;
}
```

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

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

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

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

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

Incorrect Input and the Error State of the Input Stream

When your program requests input through the input operator and the input provided is incorrect or of the wrong type, the error state may be set in the input stream and further input from that input stream may fail. One runtime symptom of such a failure is that your program's prompts for further input display without pausing to wait for the input.

Display Output on Standard Output or Standard Error

The USL I/O Stream Library predefines three output streams, as well as the cin input stream. The standard output stream is cout, and the remaining streams, cerr and clog, are standard error streams. Output to cout goes to the C standard output stream, stdout, unless cout has been redirected. Output to cerr and clog goes to the C standard error stream, stderr, unless cerr or clog has been redirected.

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

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

```
#include <iostream.h>
int main(int argc, char* argv[]) {
   if (argc==1) cout << "Good day!" << endl;
   else cerr << "I don't know what to do with "
             << argv[1] << endl;
   return 0;
}
```

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

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

An output operator must exist for any type being output. The USL I/O Stream Library defines output operators for all C++ built-in types. For types you define yourself, you need to provide your own output operators. If you attempt to place the contents of a variable into an output stream and no output operator is defined for the type of that variable, the compiler displays an error message with text similar to the following:

```
The call does not match any parameter list for "operator<<".
```

Multiple Variables in an Output Statement

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

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

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

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

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

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

Use Output Streams other than cout, cerr, and clog

You can use the same techniques for output to other output streams as for output to the predefined output streams. The only difference is that, for other output streams, your program must define the stream. Assuming you have defined a stream attached to a file opened for output, and have called that stream myout, you can write to that file through its stream, by specifying the stream's name instead of cout, cerr or clog:

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

Flush Output Streams with endl and flush

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

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

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

A stream is flushed implicitly in the following situations:

- The predefined streams cout and clog are flushed when input is requested from the predefined input stream (cin).
- The predefined stream cerr is flushed after each output operation.
- An output stream that is unit-buffered is flushed after each output operation. A unit-buffered stream is a stream that has ios::unitbuf set.
- An output stream is flushed whenever the flush() member function is applied to it. This includes cases where the flush or endl manipulators are written to the output stream.
- The program terminates.

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

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

```
cout.flush();
secondVeryLongCalc();
cout << "All done!"
cout.flush();
```

Placing endl or flush in an Output Stream

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

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

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

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

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

is not as efficient as:

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

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

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

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

Parse Multiple Inputs

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

```
#include <iostream.h>
int main(int argc, char *argv[]) {
   int a,b,c;
   cin >> a >> b >> c;
   cout << "a: <" << a << ">\n"
        << "b: <" << b << ">\n"
        << "c: <" << c << '>' << endl;
   return 0;
}
```

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

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

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.

   400 ILE C++ provides overloads of the fstream and istream constructors and their open functions, which allow you to specify the ccsid of a file.
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.

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.
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.

Combine Input and Output of Different Types

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

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

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

Advanced USL I/O Stream Tasks

Associate a File with a Standard Input or Output Stream

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

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

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

#include <iostream.h>
#include <fstream.h>

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

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

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

```
            outfile->close();           // then close it
            }
         }
         return 0;
      }
```

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

Move through a file with filebuf Functions

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

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

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

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

int main(int argc, char *argv[]) {
   // declare a streampos object to keep track of the position in filebuf
   streampos Position;

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

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

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

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

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

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

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

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

   // Now close the file and end the program
   InputFile.close();
   return 0;
   }
```

If the file algonq.uin contains the following text:

```
The trip begins on Round Lake.
We proceed through a marshy portage,
and soon find ourselves in a river whose water is the color of ink.
A heron flies off in the distance.
Frogs croak cautiously alongside the canoes.
We can feel the sun's heat glaring at us from grassy shores.
```

the output of the example program is:

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

Use Encoded and Relative Byte Offsets to Move through a File

The following example shows how you can use both encoded and relative byte offsets to move through a file. Note that encoded offsets are specific to z/OS C/C++ and programs that use them may not be portable.

```
// Example of using encoded and relative byte offsets
// in seeking through a file

#include <iomanip.h>
#include <fstream.h>

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

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

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

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

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

   for (j = 0; j < 4; ++j) {
      (*fb).seekoff(off[j], ios::beg, ios::in);
      fs >> i;
      cout << "data at pos" << dec << setfill(' ') << setw(4) << pos[j]
           << " is \"" << setw(4) << i << "\" (encoded offset was 0x"
```

```
            << hex << setfill('0') << setw(8) << off[h] << ")" << endl;
      if (i != pos[j]) return 37 + 10*j;
   }
   fs.close();
   cout.fill(' ');
   cout.setf(ios::dec, ios::basefield);
   cout << "\nOpen the file in binary byteseek mode, reposition using\n"
        << "byte offsets calculated by the user program" << endl;
   fs.open("tseek.data", "byteseek", ios::in|ios::binary);
   fb = fs.rdbuf();
   for (j = 0, j < 4; ++j) {
      off[j] = (*fb).seekoff(4*pos[j], ios::beg, ios::in);
      fs >> i;
      cout << "data at pos" << setw(4) << pos[j] << "is \"" << setw(4) << i
           << "\" (byte    offset was " << setw(10) << off[j] << ")" << endl;
      if (i != pos[j]) return 77 + 10*j;
    }
    return 0;
}
```

Define an Input Operator for a Class Type

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

myclass.h

```
#include <iostream.h>

class PhoneNumber {
   public:
      int AreaCode;
      int Exchange;
      int Local;
// Copy Constructor:
      PhoneNumber(int ac, int ex, int lc) :
         AreaCode(ac), Exchange(ex), Local(lc) {}
//... Other member functions
};
istream& operator>> (istream& aStream, PhoneNumber& aPhoneNum) {
   int tmpAreaCode, tmpExchange, tmpLocal;
   aStream >> tmpAreaCode >> tmpExchange >> tmpLocal;
   aPhoneNum=PhoneNumber(tmpAreaCode, tmpExchange, tmpLocal);
   return aStream;
   }
```

The input operator must have the following characteristics:

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

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

Use the cin Stream in a Class Input Operator

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

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

is rewritten as:

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

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

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

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

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

Display Prompts in Input Operator Code

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

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

You may be tempted to do this when you anticipate that the source of all input for objects of a class will be the standard input stream cin. Avoid this practice wherever possible, because a program using your class may later attempt to read input into an object of your class from a different stream (for example, an fstream object attached to a file). In such cases, the prompts are still written to cout even

though input from cin is not consumed by the input operation. Such an interface does not prevent programs from using your class, but the unnecessary prompts may puzzle end users.

Use Output Streams Other than cout, cerr, and clog

You can use the same techniques for output to other output streams as for output to the predefined output streams. The only difference is that, for other output streams, your program must define the stream. Assuming you have defined a stream attached to a file opened for output, and have called that stream myout, you can write to that file through its stream, by specifying the stream's name instead of cout, cerr or clog:

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

Define an Output Operator for a Class Type

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

myclass.h

```
#include <iostream.h>

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

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

The output operator must have the following characteristics:

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

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

Class Output Operators and the Format State

You should consider checking the state of applicable format flags for any stream you perform output to in a class output operator. At the very least, if you change

the format state in your class output operator, before your operator returns it should reset the format state to what it was on entry to the operator. For example, if you design an output operator to always write floating-point numbers at a given precision, you should save the precision in a temporary on entry to your operator, then change the precision and do your output, and reset the precision before returning.

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

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

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

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

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

int main(int argc, char *argv[]) {
   Coord_3D MyCoord(38.162168,1773.59,17293.12);
   cout << setw(17) << MyCoord << '\n'
        << setw(11) << MyCoord << endl;
   return 0;
   }
```

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

```
      38.16221773.5917293.1
38.16221773.5917293.1
```

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

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

Correct Input Stream Errors

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

```
#include <iostream.h>
int main(int argc, char *argv[])
   {
   int i=-1;
   while (i<=0)
   {
      cout << "Enter a positive integer: " ;
      cin >> i;
   }
   cout << "The value was " << i << endl;
   return 0;
   }
```

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

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

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

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

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

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

```
#include <iostream.h>

int main(int argc, char *argv[]) {
   int i=-1;
   while (i==-1) {
      cout << "Enter an integer value: ";
      cin >> i;
      while (cin.fail()) {   // could also be "while (!cin) {"
         cin.clear();
         cin.ignore(1000,'\n');
         cerr << "Please try again: ";
         cin >> i;
        }
```

```
      }
   cout << "The value was " << i << endl;
   return 0;
}
```

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

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

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

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

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

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

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

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

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.

For new applications, you may want to consider using IString or IText rather than strstream to handle strings. These classes provides a much broader range of string-handling capabilities than strstream, including the ability to use mathematical operators such as + (to concatenate two strings), = (to copy one string to another), and == (to compare two strings for equality).

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.
```

Chapter 2. USL Complex Mathematics Library

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.

Review of Complex Numbers

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

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

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*
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

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)
   };
```

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.

Use Mathematical Operators for complex

With these operators, you can code expressions on complex numbers such as the expressions shown in the example below. In the example, for each complex scalar x, the comments showing the results of operations use xr to denote the scalar's real part and xi to denote the scalar's imaginary part.

```
//   Using the complex mathematical operators

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

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

int main(int argc, char *argv[])
{
   cout << "Enter six complex numbers, separated by spaces:\n";
   cin >> b >> c >> d >> e >> f >> g;

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

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

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

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

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

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

```
        << "f=" << f << '\n'
        << "g=" << g << endl;
   return 0;
}
```

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

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

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

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)

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)
```

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.

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
```

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().

Handle complex Mathematics Errors

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

- use the c_exception class
- z/OS AIX define a customized complex_error function
- z/OS AIX 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.

z/OS AIX Define a Customized complex_error Function

You can either use the default version of complex_error() or define your own version of the function. 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)
```

z/OS AIX 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.

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(2*pi* / *n*), sin(2*pi* / *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)
```

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     ~     ~    !=    ==
```

Chapter 3. Reference

_CCSID_T

400 This class is specific to the OS/400 implementation. Its use will lead to nonportable code.

The C++ Standard Library and the USL Library use this class to pass Coded Character Set ID (CCSID) information to the streaming functions. There are two identical versions of this class, one for the C++ Standard Library in the std namespace and the other for the USL Library in the global namespace.

Class header file: fstream.h

_CCSID_T - Hierarchy List

_CCSID_T

_CCSID_T - Member Functions and Data by Group

Constructors & Destructor

_CCSID_T

```
public:_CCSID_T(int ii)
```

This is supported on 400

Constructs an object of this class. The int parameter represents an OS/400 numeric CCSID.

Query Functions

value

```
public:int value() const
```

This is supported on 400

Returns the OS/400 numeric Coded Character Set IDentifier (CCSID).

_CCSID_T - Inherited Member Functions and Data

Inherited Public Functions

None

Inherited Public Data

None

Inherited Protected Functions

None

Inherited Protected Data

None

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)
```

This is supported on AIX 400 z/OS

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()
```

This is supported on AIX 400 z/OS

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&)
```

This is supported on AIX

Overload 2

```
public:inline void operator *=(complex)
```

This is supported on 400 z/OS

operator +=

Assigns the value of x + y to x.

Overload 1

```
public:inline void operator +=(complex)
```

This is supported on 400 z/OS

Overload 2

```
public:inline void operator +=(const complex&)
```

This is supported on AIX

operator -=

Assigns the value of x - y to x.

Overload 1

```
public:inline void operator -=(complex)
```

This is supported on 400 z/OS

Overload 2

```
public:inline void operator -=(const complex&)
```

This is supported on AIX

operator /=

Assigns the value of x / y to x.

Overload 1

```
public:inline void operator /=(complex)
```

This is supported on 400 z/OS

Overload 2

```
public:void operator /=(const complex&)
```

This is supported on AIX

Internal Functions

These functions are internal to the complex class and should not be used by application programs.

hexdiveq

```
public:void hexdiveq(complex)
```

This is supported on z/OS

An internal function called by operator/= when the application uses hexadecimal floating point and double values.

hexmuteq

```
public:void hexmuteq(complex)
```

This is supported on z/OS

An internal function called by operator*= when the application uses hexadecimal floating point and double values.

ieeediveq

```
public:void ieeediveq(complex)
```

This is supported on z/OS

An internal function called by operator/= when the application uses IEEE floating point and double values.

ieeemuteq

```
public:void ieeemuteq(complex)
```

This is supported on z/OS

An internal function called by operator*= when the application uses IEEE floating point and double values.

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 $a^2 + b^2$.

This is supported on 400 z/OS

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 $a^2 + b^2$.

This is supported on AIX

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.

This is supported on 400 z/OS

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.

This is supported on AIX

conj

```
complex conj(complex)
```

Returns the complex value equal to (a, -b) if the input argument is equal to (a, b).

This is supported on 400 z/OS

conj

```
inline complex conj(const complex&)
```

Returns the complex value equal to (a, -b) if the input argument is equal to (a, b).

This is supported on AIX

cos

```
complex cos(complex)
```

Returns the cosine of the complex argument.

This is supported on 400 z/OS

cos

```
complex cos(const complex&)
```

Returns the cosine of the complex argument.

This is supported on AIX

cosh

```
complex cosh(complex)
```

Returns the hyperbolic cosine of the complex argument.

This is supported on 400 z/OS

cosh

```
complex cosh(const complex&)
```

Returns the hyperbolic cosine of the complex argument.

This is supported on AIX

exp

```
complex exp(complex)
```

Returns the complex value equal to e to the power of x where x is the argument.

This is supported on 400 z/OS

exp

```
complex exp(const complex&)
```

Returns the complex value equal to e to the power of x where x is the argument.

This is supported on AIX

imag

```
double imag(const complex&)
```

Extracts the imaginary part of the complex number provided as the argument.

This is supported on 400 z/OS

imag

```
inline double imag(const complex&)
```

Extracts the imaginary part of the complex number provided as the argument.

This is supported on AIX

log

```
complex log(complex)
```

Returns the natural logarithm of the argument x.

This is supported on AIX

log

```
complex log(complex)
```

Returns the natural logarithm of the argument x.

This is supported on 400 z/OS

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.

This is supported on 400 z/OS

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.

This is supported on AIX

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.

This is supported on 400 z/OS

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.

This is supported on AIX

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.

This is supported on 400 z/OS

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.

This is supported on AIX

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.

This is supported on AIX

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.

This is supported on 400 z/OS

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.

This is supported on AIX

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.

This is supported on AIX

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.

This is supported on 400 z/OS

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.

This is supported on AIX

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.

This is supported on 400 z/OS

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.

This is supported on AIX

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.

This is supported on AIX

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.

This is supported on 400 z/OS

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.

This is supported on 400 z/OS

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.

This is supported on AIX

polar

```
complex polar(double, double = 0)
```

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

This is supported on AIX 400 z/OS

pow

```
complex pow(complex, double)
```

Returns the complex value x^y, where x is the first argument and y is the second argument.

This is supported on 400 z/OS

pow

```
complex pow(double, complex)
```

Returns the complex value x^y, where x is the first argument and y is the second argument.

This is supported on 400 z/OS

pow

```
complex pow(complex, complex)
```

Returns the complex value x^y, where x is the first argument and y is the second argument.

This is supported on 400 z/OS

pow

```
complex pow(complex, int)
```

Returns the complex value x^y, where x is the first argument and y is the second argument.

This is supported on 400 z/OS

pow

```
complex pow(const complex&, int)
```

Returns the complex value x^y, where x is the first argument and y is the second argument.

This is supported on AIX

pow

```
complex pow(const complex&, double)
```

Returns the complex value x^y, where x is the first argument and y is the second argument.

This is supported on AIX

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.

This is supported on AIX

pow

```
complex pow(double, const complex&)
```

Returns the complex value x^y, where x is the first argument and y is the second argument.

This is supported on AIX

real

```
double real(const complex&)
```

Extracts the real part of the complex number provided as the argument.

This is supported on 400 z/OS

real

```
inline double real(const complex&)
```

Extracts the real part of the complex number provided as the argument.

This is supported on AIX

sin

```
complex sin(const complex&)
```

Returns the sine of the complex argument.

This is supported on AIX

sin

```
complex sin(complex)
```

Returns the sine of the complex argument.

This is supported on 400 z/OS

sinh

```
complex sinh(const complex&)
```

Returns the hyperbolic sine of the complex argument.

This is supported on AIX

sinh

```
complex sinh(complex)
```

Returns the hyperbolic sine of the complex argument.

This is supported on 400 z/OS

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.

This is supported on AIX 400 z/OS

complex - 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()
```

This is supported on AIX 400 z/OS

The filebuf destructor calls filebuf.close().

filebuf

Overload 1

```
public:filebuf(int fd, char* p, long l)
```

This is supported on AIX z/OS

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.

This function is available for 64-bit applications. The third argument is a long value.

Overload 2

```
public:filebuf(int fd)
```

This is supported on AIX 400 z/OS

Constructs a filebuf object that is attached to the file descriptor fd.

Overload 3

```
public:filebuf(int fd, char* p, int l)
```

This is supported on AIX 400 z/OS

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 and z/OS Considerations

This function is available for 32-bit applications. The third argument is an int value.

Overload 4

```
public:filebuf()
```

This is supported on AIX 400 z/OS

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(FILE* fp)
```

This is supported on z/OS

Attaches the filebuf object to the file pointer fp. If the filebuf object is already open, attach() returns 0. Otherwise, attach() returns a pointer to the filebuf object.

z/OS Considerations

If you have a file pointer already opened, use this function to do the attach instead of using the file descriptor.

Overload 2

```
public:filebuf* attach(int fd)
```

This is supported on AIX 400 z/OS

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()
```

This is supported on AIX 400 z/OS

Returns a nonzero value if the filebuf object is attached to a file descriptor. Otherwise, is_open() returns zero.

Data members

openprot

```
public:static const int openprot
```

This is supported on AIX 400 z/OS

The default protection mode used when opening files.

in_start

```
protected:char* in_start
```

This is supported on AIX 400 z/OS

Data member.

lahead

```
protected:char lahead [ 2 ]
```

This is supported on AIX 400 z/OS

A variable used to store look-ahead characters during underflow processing.

last_seek

```
protected:streampos last_seek
```

This is supported on AIX 400 z/OS

Stream position last seeked to.

mode

```
protected:int mode
```

This is supported on AIX 400 z/OS

Open mode of the filebuf object.

opened

```
protected:char opened
```

This is supported on AIX 400 z/OS

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
```

This is supported on AIX 400 z/OS

The file descriptor of the file attached to the filebuf object.

Detach Functions

close

```
public:filebuf* close()
```

This is supported on AIX 400 z/OS

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()
```

This is supported on AIX 400 z/OS

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

fp

```
public:FILE* fp()
```

This is supported on z/OS

Returns the file pointer that is attached to the filebuf object. If the filebuf object is not opened, fp() returns 0.

overflow

```
public:virtual int overflow(int = EOF)
```

This is supported on AIX 400 z/OS

Emptys an output buffer. Returns EOF when an error occurs. Returns 0 otherwise.

seekoff

```
public:virtual streampos seekoff(streamoff, ios::seek_dir, int)
```

This is supported on AIX 400 z/OS

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.

z/OS Considerations

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

- The file is not a variable record format file, and is opened for binary I/O.
- The file is a variable record format file, and is opened for binary I/O with the byteseek option. The byteseek option is enabled for a specific file if the byteseek fopen() option is passed when the file is opened. The byteseek option can also be enabled for all files if you set the _EDC_BYTESEEK environment variable.

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

sync

```
public:virtual int sync()
```

This is supported on 

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()
```

This is supported on 

Fills an input buffer. Returns EOF when an error occurs or the end of the input is reached. Returns the next character otherwise.

Open Functions

z/OS Considerations

The prot parameter is ignored.

open

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.

Overload 1

```
public:filebuf*
  open( const char* name,
        int om,
        int prot = openprot )
```

This is supported on AIX z/OS

Overload 2

```
public:filebuf*
  open( const char* name,
        int om,
        int prot = openprot,
        _CCSID_T ccsid = _CCSID_T ( 0 ) )
```

This is supported on 400

Overload 3

```
public:filebuf*
  open( const char* name,
        const char* attr,
        int om,
        int prot = openprot )
```

This is supported on z/OS

You can use the attr parameter to specify additional file attributes, such as lrecl or recfm. All the parameters documented for the fopen() function are supported, with the exception of type=record.

Overload 4

```
public:filebuf* open(const char* name, int om, _CCSID_T ccsid)
```

This is supported on 400

Query Functions

fd

```
public:int fd()
```

This is supported on AIX 400 z/OS

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()
```

This is supported on AIX 400 z/OS

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)
```

This is supported on AIX z/OS

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.

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)
```

This is supported on AIX 400 z/OS

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 and z/OS 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**	**Definition**	**Page Number**
~streambuf	175	dbp	178
in_avail	176	optim_in_avail	176
optim_sbumpc	176	out_waiting	183
overflow	183	pbackfail	183
pptr_non_null	179	sbumpc	176
seekoff	179	seekpos	179
setbuf	185	sgetc	176
sgetn	177	snextc	177
sputbackc	184	sputc	184
sputn	184	stossc	180
streambuf	175	streambuf_resource	185
xsgetn	178	xsputn	185

Inherited Public Data

None

Inherited Protected Functions

streambuf			
Definition	**Page Number**	**Definition**	**Page Number**
allocate	187	base	180
blen	187	doallocate	188
eback	180	ebuf	180
egptr	180	epptr	180
gbump	181	gptr	181
pbase	181	pbump	181
pptr	182	setb	182
setg	182	setp	182
unbuffered	188		

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()
```

This is supported on AIX 400 z/OS

Destructs an fstream object.

fstream

Constructs an object of this class.

Overload 1

```
public:fstream(int fd, char* p, int l)
```

This is supported on AIX 400 z/OS

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 and z/OS Considerations

This function is available for 32-bit applications. The third argument is an int value.

Overload 2

```
public:fstream(const char* name, int mode, _CCSID_T ccsid)
```

This is supported on 400

Constructs an fstream object and opens the file name with open mode equal to mode and ccsid equal to ccsid.

If the file cannot be opened, the error state of the constructed fstream object is set.

If the ccsid parameter is non-zero then it is treated as a CCSID (coded character set identifier) and will correspond to the CCSID of data written to and from the file. If the parameter value is zero then the CCSID of the job will be used.

Overload 3

```
public:fstream(int fd)
```

This is supported on AIX 400 z/OS

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 4

```
public:fstream(int fd, char* p, long l)
```

This is supported on AIX z/OS

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.

This function is available for 64-bit applications. The third argument is a long value.

Overload 5

```
public:fstream( const char* name,
         int mode,
         int prot = filebuf::openprot,
         _CCSID_T ccsid = _CCSID_T ( 0 ) )
```

This is supported on 400

Constructs an fstream object and opens the file name with open mode equal to mode and protection mode equal to prot, and ccsid equal to ccsid.

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.

If the ccsid parameter is non-zero then it is treated as a CCSID (coded character set identifier) and will correspond to the CCSID of data written to and from the file. If the parameter value is zero then the CCSID of the job will be used.

Overload 6

```
public:fstream( const char* name,
         const char* attr,
         int mode,
         int prot = filebuf::openprot )
```

This is supported on z/OS

Constructs an fstream object and opens the file name with open mode equal to mode, attributes equal to attr 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.

You can use the attr parameter to specify additional file attributes such as lrecl or recfm. All the parameters documented for the fopen() functions are supported, with the exception of type=record.

z/OS Considerations

The prot attribute is ignored.

Overload 7

```
public:fstream( const char* name,
         int mode,
         int prot = filebuf::openprot )
```

This is supported on AIX z/OS

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.

z/OS Considerations

The prot attribute is ignored.

Overload 8

```
public:fstream()
```

This is supported on AIX 400 z/OS

Constructs an unopened fstream object.

Filebuf Functions

Use these functions to work with the underlying filebuf object.

rdbuf

```
public:filebuf* rdbuf()
```

This is supported on AIX 400 z/OS

Returns a pointer to the filebuf object that is attached to the fstream object.

Open Functions

Opens the file.

z/OS Considerations

The prot parameter is ignored.

open

Opens the specified file.

Overload 1

```
public:void
  open( const char* name,
        int mode,
        int prot = filebuf::openprot )
```

This is supported on AIX z/OS

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.

Overload 2

```
public:void
  open( const char* name,
        int mode,
        int prot = filebuf::openprot,
        _CCSID_T ccsid = _CCSID_T ( 0 ) )
```

This is supported on 400

Opens the file with the specified name, mode, protection and coded character set id 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 or 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.

If the ccsid parameter is non-zero then it is treated as a CCSID (coded character set identifier) and will correspond to the CCSID of data written to and from the file. If the parameter value is zero then the CCSID of the job will be used.

Overload 3

```
public:void
  open( const char* name,
        const char* attr,
        int mode,
        int prot = filebuf::openprot )
```

This is supported on z/OS

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.

You can use the attr parameter to specify additional file attributes, such as lrecl or recfm. All the parameters documented for the fopen() function are supported, with the exception of type=record.

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.

Overload 4

```
public:void open(const char* name, int mode, _CCSID_T ccsid)
```

This is supported on 400

Opens the file with the specified name, mode and coded character set id and attaches it to the fstream object.

If the file with the name, name does not already exist, open() tries to create it unless ios::nocreate is set.

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.

If the ccsid parameter is non-zero then it is treated as a CCSID (coded character set identifier) and will correspond to the CCSID of data written to and from the file. If the parameter value is zero then the CCSID of the job will be used.

fstream - Inherited Member Functions and Data

Inherited Public Functions

fstreambase			
Definition	**Page Number**	**Definition**	**Page Number**
~fstreambase	79	attach	81
close	82	detach	82
fstreambase	79	setbuf	84

ios			
Definition	**Page Number**	**Definition**	**Page Number**
~ios	92	bad	94
bitalloc	99	clear	94
eof	94	fail	94
fill	95	flags	95
good	94	ios	92
ios_resource	98	iword	99
operator !	99	operator const void *	99
operator void *	99	precision	95
pword	99	rdbuf	99
rdstate	94	setf	96
skip	97	sync_with_stdio	100
tie	100	unsetf	97
width	97	xalloc	101

Inherited Public Data

ios			
Definition	**Page Number**	**Definition**	**Page Number**
adjustfield	92	basefield	92
floatfield	92		

Inherited Protected Functions

ios			
Definition	**Page Number**	**Definition**	**Page Number**
init	98	ios	92
setstate	95		

fstreambase			
Definition	**Page Number**	**Definition**	**Page Number**
verify	82		

Inherited Protected Data

ios			
Definition	**Page Number**	**Definition**	**Page Number**
(* stdioflush) ( )	101	assign_private	92
bp	93	delbuf	93
isfx_special	93	ispecial	93
osfx_special	93	ospecial	93
state	93	x_fill	98

ios			
Definition	**Page Number**	**Definition**	**Page Number**
x_flags	93	x_precision	98
x_tie	93	x_width	98

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()
```

This is supported on 

Destructs an fstreambase object.

fstreambase

Constructs an object of this class.

Overload 1

```
public:fstreambase(int fd, char* p, int l)
```

This is supported on 

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 and z/OS Considerations

This function is available for 32-bit applications. The third argument is an int value.

Overload 2

```
public:fstreambase( const char* name,
                    const char* attr,
                    int mode,
                    int prot = filebuf::openprot )
```

This is supported on z/OS

Constructs an fstreambase object, initializes the ios object, and opens the specified file with the specified mode, attributes and protection.

You can use the attr parameter to specify additional file attributes such as lrecl or recfm. All the parameters documented for the fopen() functions are supported, with the exception of type=record.

z/OS Considerations

The prot parameter is ignored.

Overload 3

```
public:fstreambase(int fd)
```

This is supported on AIX 400 z/OS

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 4

```
public:fstreambase(int fd, char* p, long l)
```

This is supported on AIX z/OS

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.

This function is available for 64-bit applications. The third argument is a long value.

Overload 5

```
public:fstreambase( const char* name,
                    int mode,
                    int prot = filebuf::openprot,
                    _CCSID_T ccsid = _CCSID_T ( 0 ) )
```

This is supported on 400

Constructs an fstreambase object, initializes the ios object, and opens the specified file with the specified mode, protection, and ccsid.

If the ccsid parameter is non-zero then it is treated as a CCSID (coded character set identifier) and will correspond to the CCSID of data written to and from the file. If the parameter value is zero then the CCSID of the job will be used.

Overload 6

```
public:fstreambase(const char* name, int mode, _CCSID_T ccsid)
```

This is supported on 400

Constructs an fstreambase object, initializes the ios object, and opens the specified file with the specified mode and ccsid.

If the ccsid parameter is non-zero then it is treated as a CCSID (coded character set identifier) and will correspond to the CCSID of data written to and from the file. If the parameter value is zero then the CCSID of the job will be used.

Overload 7

```
public:fstreambase( const char* name,
                    int mode,
                    int prot = filebuf::openprot )
```

This is supported on AIX z/OS

Constructs an fstreambase object, initializes the ios object, and opens the specified file with the specified mode and protection.

z/OS Considerations

The prot parameter is ignored.

Overload 8

```
public:fstreambase()
```

This is supported on AIX 400 z/OS

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(FILE* fp)
```

This is supported on z/OS

Attaches the fstream, ifstream or ofstream object to the file pointer fp. If the object is already attached to a file pointer, an error occurs and ios::failbit is set in the format state of the object.

Overload 2

```
public:void attach(int fd)
```

This is supported on AIX 400 z/OS

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()
```

This is supported on AIX 400 z/OS

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()
```

This is supported on AIX 400 z/OS

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()
```

This is supported on AIX 400 z/OS

Returns a pointer to the filebuf object that is attached to the fstream, ifstream or ofstream object.

Miscellaneous Functions

verify

```
protected:void verify(int)
```

This is supported on AIX 400 z/OS

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,
        const char* attr,
        int mode,
        int prot = filebuf::openprot )
```

This is supported on z/OS

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.

You can use the attr parameter to specify additional file attributes, such as lrecl or recfm. All the parameters documented for the fopen() function are supported, with the exception of type=record.

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.

Overload 2

```
public:void open(const char* name, int mode, _CCSID_T ccsid)
```

This is supported on 400

Opens the file with the specified name, mode and coded character set id and attaches it to the fstream object.

If the file with the name, name does not already exist, open() tries to create it unless ios::nocreate is set.

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.

If the ccsid parameter is non-zero then it is treated as a CCSID (coded character set identifier) and will correspond to the CCSID of data written to and from the file. If the parameter value is zero then the CCSID of the job will be used.

Overload 3

```
public:void
  open( const char* name,
        int mode,
        int prot = filebuf::openprot,
        _CCSID_T ccsid = _CCSID_T ( 0 ) )
```

This is supported on 400

Opens the file with the specified name, mode, protection and coded character set id 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 or 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.

If the ccsid parameter is non-zero then it is treated as a CCSID (coded character set identifier) and will correspond to the CCSID

of data written to and from the file. If the parameter value is zero then the CCSID of the job will be used.

Overload 4

```
public:void
  open( const char* name,
        int mode,
        int prot = filebuf::openprot )
```

This is supported on AIX z/OS

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)
```

This is supported on AIX z/OS

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.

This function is available for 64-bit applications. The second argument is a long value.

Overload 2

```
public:void setbuf(char* p, int l)
```

This is supported on AIX 400 z/OS

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 and z/OS 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**	**Definition**	**Page Number**
~ios	92	bad	94
bitalloc	99	clear	94
eof	94	fail	94
fill	95	flags	95
good	94	ios	92
ios_resource	98	iword	99
operator !	99	operator const void *	99
operator void *	99	precision	95
pword	99	rdbuf	99
rdstate	94	setf	96
skip	97	sync_with_stdio	100
tie	100	unsetf	97
width	97	xalloc	101

Inherited Public Data

ios			
Definition	**Page Number**	**Definition**	**Page Number**
adjustfield	92	basefield	92
floatfield	92		

Inherited Protected Functions

ios			
Definition	**Page Number**	**Definition**	**Page Number**
init	98	ios	92
setstate	95		

Inherited Protected Data

ios			
Definition	**Page Number**	**Definition**	**Page Number**
(* stdioflush) ( )	101	assign_private	92
bp	93	delbuf	93
isfx_special	93	ispecial	93
osfx_special	93	ospecial	93
state	93	x_fill	98
x_flags	93	x_precision	98
x_tie	93	x_width	98

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()
```

This is supported on AIX 400 z/OS

Destructs an ifstream object.

ifstream

Constructs an object of this class.

Overload 1

```
public:ifstream(int fd, char* p, int l)
```

This is supported on AIX 400 z/OS

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 and z/OS Considerations

This function is available for 32-bit applications. The third argument is an int value.

Overload 2

```
public:ifstream( const char* name,
          const char* attr,
          int mode = ios::in,
          int prot = filebuf::openprot )
```

This is supported on z/OS

Constructs an ifstream object and opens the file name with open mode equal to mode, attributes equal to attr 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.

You can use the attr parameter to specify additional file attributes such as lrecl or recfm. All the parameters documented for the fopen() functions are supported, with the exception of type=record.

z/OS Considerations

The prot attribute is ignored.

Overload 3

```
public:ifstream(int fd)
```

This is supported on AIX 400 z/OS

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 4

```
public:ifstream(int fd, char* p, long l)
```

This is supported on AIX z/OS

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.

This function is available for 64-bit applications. The third argument is a long value.

Overload 5

```
public:ifstream( const char* name,
          int mode = ios::in,
          int prot = filebuf::openprot,
          _CCSID_T ccsid = _CCSID_T ( 0 ) )
```

This is supported on 400

Constructs an ifstream object and opens the file name with open mode equal to mode and protection mode equal to prot, and ccsid equal to ccsid. 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.

If the ccsid parameter is non-zero then it is treated as a CCSID (coded character set identifier) and will correspond to the CCSID of data written to and from the file. If the parameter value is zero then the CCSID of the job will be used.

Overload 6

```
public:ifstream(const char* name, int mode, _CCSID_T ccsid)
```

This is supported on 400

Constructs an ifstream object and opens the file name with open mode equal to mode and ccsid equal to ccsid. If the file cannot be opened, the error state of the constructed fstream object is set.

If the ccsid parameter is non-zero then it is treated as a CCSID (coded character set identifier) and will correspond to the CCSID of data written to and from the file. If the parameter value is zero then the CCSID of the job will be used.

Overload 7

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

This is supported on AIX z/OS

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.

z/OS Considerations

The prot attribute is ignored.

Overload 8

```
public:ifstream()
```

This is supported on AIX 400 z/OS

Constructs an unopened ifstream object.

Filebuf Functions

rdbuf

```
public:filebuf* rdbuf()
```

This is supported on AIX 400 z/OS

Returns a pointer to the filebuf object that is attached to the ifstream object.

Open Functions

Opens the file.

z/OS Considerations

The prot attribute is ignored.

open

Opens the specified file.

Overload 1

```
public:void
  open( const char* name,
        int mode = ios::in,
        int prot = filebuf::openprot,
        _CCSID_T ccsid = _CCSID_T ( 0 ) )
```

This is supported on 400

Opens the file with the specified name, mode, protection and coded character set id 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 or 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.

If the ccsid parameter is non-zero then it is treated as a CCSID (coded character set identifier) and will correspond to the CCSID of data written to and from the file. If the parameter value is zero then the CCSID of the job will be used.

Overload 2

```
public:void
  open( const char* name,
        int mode = ios::in,
        int prot = filebuf::openprot )
```

This is supported on AIX z/OS

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.

Overload 3

```
public:void open(const char* name, int mode, _CCSID_T ccsid)
```

This is supported on 400

Opens the file with the specified name, mode and coded character set id and attaches it to the fstream object.

If the file with the name, name does not already exist, open() tries to create it unless ios::nocreate is set.

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.

If the ccsid parameter is non-zero then it is treated as a CCSID (coded character set identifier) and will correspond to the CCSID of data written to and from the file. If the parameter value is zero then the CCSID of the job will be used.

Overload 4

```
public:void
  open( const char* name,
        const char* attr,
        int mode = ios::in,
        int prot = filebuf::openprot )
```

This is supported on z/OS

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.

You can use the attr parameter to specify additional file attributes, such as lrecl or recfm. All the parameters documented for the fopen() function are supported, with the exception of type=record.

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**	**Definition**	**Page Number**
~fstreambase	79	attach	81
close	82	detach	82
fstreambase	79	open	82
setbuf	84		

ios			
Definition	**Page Number**	**Definition**	**Page Number**
~ios	92	bad	94
bitalloc	99	clear	94
eof	94	fail	94
fill	95	flags	95
good	94	ios	92
ios_resource	98	iword	99
operator !	99	operator const void *	99
operator void *	99	precision	95
pword	99	rdbuf	99
rdstate	94	setf	96
skip	97	sync_with_stdio	100
tie	100	unsetf	97
width	97	xalloc	101

Inherited Public Data

ios			
Definition	**Page Number**	**Definition**	**Page Number**
adjustfield	92	basefield	92
floatfield	92		

Inherited Protected Functions

ios			
Definition	**Page Number**	**Definition**	**Page Number**
`init` `setstate`	98 95	`ios`	92

fstreambase			
Definition	**Page Number**	**Definition**	**Page Number**
`verify`	82		

Inherited Protected Data

ios			
Definition	**Page Number**	**Definition**	**Page Number**
`( * stdioflush ) ( )`	101	`assign_private`	92
`bp`	93	`delbuf`	93
`isfx_special`	93	`ispecial`	93
`osfx_special`	93	`ospecial`	93
`state`	93	`x_fill`	98
`x_flags`	93	`x_precision`	98
`x_tie`	93	`x_width`	98

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()
```

This is supported on AIX 400 z/OS

Destructs an ios object.

ios

Creates an ios object.

Overload 1

```
public:ios(streambuf*)
```

This is supported on AIX 400 z/OS

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 is supported on AIX 400 z/OS

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
```

This is supported on AIX 400 z/OS

Data member for the ios class.

basefield

```
public:static const long basefield
```

This is supported on AIX 400 z/OS

Data member for the ios class.

floatfield

```
public:static const long floatfield
```

This is supported on AIX 400 z/OS

Data member for the ios class.

assign_private

```
protected:int assign_private
```

This is supported on AIX 400 z/OS

Data member for the ios class.

bp

```
protected:streambuf* bp
```

This is supported on AIX 400 z/OS

Data member for the ios class. Pointer to the streambuf object.

delbuf

```
protected:int delbuf
```

This is supported on AIX 400 z/OS

Data member for the ios class.

isfx_special

```
protected:int isfx_special
```

This is supported on AIX 400 z/OS

Data member for the ios class.

ispecial

```
protected:int ispecial
```

This is supported on AIX 400 z/OS

Data member for the ios class.

osfx_special

```
protected:int osfx_special
```

This is supported on AIX 400 z/OS

Data member for the ios class.

ospecial

```
protected:int ospecial
```

This is supported on AIX 400 z/OS

Data member for the ios class.

state

```
protected:int state
```

This is supported on AIX 400 z/OS

Data member for the ios class.

x_flags

```
protected:long x_flags
```

This is supported on AIX 400 z/OS

Data member for the ios class.

x_tie

```
protected:ostream* x_tie
```

This is supported on AIX 400 z/OS

Data member for the ios class.

Error State Functions

bad

```
public:int bad() const
```

This is supported on AIX 400 z/OS

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)
```

This is supported on AIX 400 z/OS

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
```

This is supported on AIX 400 z/OS

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
```

This is supported on AIX 400 z/OS

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
```

This is supported on AIX 400 z/OS

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
```

This is supported on AIX 400 z/OS

Returns the current value of the error state of the ios object.

setstate

```
protected:void setstate(int b)
```

This is supported on AIX 400 z/OS

Format State Functions

fill

Overload 1

```
public:char fill() const
```

This is supported on AIX 400 z/OS

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)
```

This is supported on AIX 400 z/OS

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
```

This is supported on AIX 400 z/OS

Returns the value of the flags that make up the current format state.

Overload 2

```
public:long flags(long f)
```

This is supported on AIX 400 z/OS

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
```

This is supported on AIX 400 z/OS

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)
```

This is supported on AIX 400 z/OS

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 is supported on AIX 400 z/OS

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 is supported on AIX 400 z/OS

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)
```

This is supported on AIX 400 z/OS

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)
```

This is supported on AIX 400 z/OS

Turns off the format flags specified in the argument and returns the previous format state.

width

Overload 1

```
public:int width() const
```

This is supported on AIX 400 z/OS

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)
```

This is supported on AIX 400 z/OS

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
```

This is supported on 

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
```

This is supported on 

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

x_width

```
protected:short x_width
```

This is supported on 

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

Initialization Functions

init

```
protected:void init(streambuf*)
```

This is supported on 

Locking functions

ios_resource

```
public:IRTLResource& ios_resource() const
```

This is supported on 

Miscellaneous Functions

bitalloc

```
public:static long bitalloc()
```

This is supported on AIX 400 z/OS

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)
```

This is supported on AIX 400 z/OS

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
```

This is supported on AIX 400 z/OS

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
```

This is supported on AIX 400 z/OS

operator void *

```
public:operator void *()
```

This is supported on AIX 400 z/OS

pword

```
public:void *& pword(int)
```

This is supported on AIX 400 z/OS

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().

On platforms where long and pointer types are the same size, pword() is the same as iword(), except that the two functions return different types.

rdbuf

```
public:streambuf* rdbuf()
```

This is supported on AIX 400 z/OS

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()
```

This is supported on AIX 400 z/OS

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()
```

This is supported on AIX 400 z/OS

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)
```

This is supported on AIX 400 z/OS

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()
```

This is supported on AIX 400 z/OS

AIX and z/OS 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()
```

This is supported on AIX z/OS

The value returned is a long for 64-bit applications. This function is not available for 32-bit applications.

(* stdioflush) ( )

```
protected:static void ( * stdioflush ) ( )
```

This is supported on AIX 400 z/OS

ios - Enumerations

White Space and Padding

The following values are set by default:

- skipws and right

skipws

If ios::skipws is set, white space will be skipped on input. If it is not set, white space is not skipped. If ios::skipws is not set, the arithmetic extractors will signal an error if you attempt to read an integer or

floating-point value that is preceded by white space. ios::failbit is set, and extraction ceases until it is cleared. This is done to avoid looping problems. If the following program is run with an input file that contains integer values separated by spaces, ios::failbit is set after the first integer value is read, and the program halts. If the program did not fail() at the beginning of the while loop to test if ios::failbit is set, it would loop indefinitely.

```
#include < fstream.h >
int main()
{
   fstream f("spadina.dat", ios::in);
   f.unsetf(ios::skipws);
   int i;
   while (!f.eof() && !f.fail()) {
      f >> i;
      cout << i;
   }
}
```

left

If ios::left is set, the value is left-justified. Fill characters are added after the value.

right

If ios::right is set, the value is right-justified. Fill characters are added before the value.

internal

If ios::internal is set, the fill characters are added after any leading sign or base notation, but before the value itself.

Base Conversion

The manipulators ios::dec, ios::oct, and ios::hex have the same effect as the flags ios::dec, ios::oct, and ios::hex respectively. dec is set by default.

dec

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

oct

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

hex

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

showbase

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

Integral Formatting

The following manipulator affects integral formatting:

showpos

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

Floating-Point Formatting

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

showpoint

If ios:showpoint is set, trailing zeros and a decimal point appear in the result of a floating-point conversion. This flag has no effect if either ios::scientific or ios::fixed is set.

scientific

If ios::scientific is set, the value is converted using scientific notation. In scientific notation, there is one digit before the decimal point and the number of digits following the decimal point depends on the value of ios::x_precision. The default value for ios::x_precision is 6. If ios::uppercase is set, an uppercase ″E″ precedes the exponent. Otherwise, a lowercase ″e″ precedes the exponent.

fixed

If ios::fixed is set, floating point values are converted to fixed notation with the number of digits after the decimal point equal to the value of ios::x_precision (or 6 by default).

If neither ios::fixed nor ios::scientific is set, the representation of floating-point values depends on their values and the number of significant digits in the representation equals ios::x_precision. Floating-point values are converted to scientific notation if the exponent resulting from a conversion to scientific notation is less an -4 or greater than or equal to the value of ios::x_precision. Otherwise, floating-point values are converted to fixed notation. If ios::showpoint is not set, trailing zeros are removed from the result and a decimal point appears only if it is followed by a digit. ios::scientific and ios::fixed are collectively identified by the static member ios::floatfield.

Uppercase and Lowercase

uppercase

If ios::uppercase is set, the operation that inserts values uses an uppercase ″E″ for floating point values in scientific notation. In addition, the operation that inserts values stores hexadecimal digits ″A″ to ″F″ in uppercase and places an uppercase ″X″ before hexadecimal values when ios::showbase is set. If ios::uppercase is not set, a lowercase ″e″ introduces the exponent in floating-point values, hexadecimal digits ″a″ to ″f″ are stored in lowercase, and a lowercase ″x″ is inserted before hexadecimal values when ios::showbase is set.

Buffer Flushing

unitbuf

If ios::unitbuf is set, ostream::osfx() performs a flush after each insertion. The attached stream buffer is unit buffered.

stdio

This flag is used internally by sync_with_stdio(). You should not use ios::stdio directly.

Variation 1

```
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 }
```

This is supported on AIX 400 z/OS

Variation 2

```
enum  { skipping=01000,
        tied=02000 }
```

This is supported on AIX 400 z/OS

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 }
```

This is supported on AIX z/OS

Variation 2

```
enum open_mode { in=1,
                 out=2,
                 ate=4,
                 app=010,
                 trunc=020,
                 nocreate=040,
                 noreplace=0100,
                 bin=0200,
                 binary=bin,
                 text=0400 }
```

This is supported on 400

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()
```

This is supported on AIX 400 z/OS

Destructs an iostream object.

iostream

Overload 1

```
public:iostream(streambuf*)
```

This is supported on AIX 400 z/OS

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()
```

This is supported on AIX 400 z/OS

Protected contructor.

iostream - Inherited Member Functions and Data

Inherited Public Functions

ios			
Definition	**Page Number**	**Definition**	**Page Number**
~ios	92	bad	94

ios			
Definition	**Page Number**	**Definition**	**Page Number**
bitalloc	99	clear	94
eof	94	fail	94
fill	95	flags	95
good	94	ios	92
ios_resource	98	iword	99
operator !	99	operator const void *	99
operator void *	99	precision	95
pword	99	rdbuf	99
rdstate	94	setf	96
skip	97	sync_with_stdio	100
tie	100	unsetf	97
width	97	xalloc	101

istream			
Definition	**Page Number**	**Definition**	**Page Number**
~istream	111	gcount	112
get	113	get_complicated	116
getline	116	ignore	119
ipfx	133	isfx	134
istream	112	operator >>	122
peek	120	putback	132
read	120	rs_complicated	121
seekg	132	sync	133
tellg	133		

Inherited Public Data

ios			
Definition	**Page Number**	**Definition**	**Page Number**
adjustfield	92	basefield	92
floatfield	92		

Inherited Protected Functions

istream			
Definition	**Page Number**	**Definition**	**Page Number**
do_ipfx	134	eatwhite	122
istream	112		

ios			
Definition	**Page Number**	**Definition**	**Page Number**
init setstate	98 95	ios	92

Inherited Protected Data

ios			
Definition	**Page Number**	**Definition**	**Page Number**
(* stdioflush) ( )	101	assign_private	92
bp	93	delbuf	93
isfx_special	93	ispecial	93
osfx_special	93	ospecial	93
state	93	x_fill	98
x_flags	93	x_precision	98
x_tie	93	x_width	98

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()
```

This is supported on AIX 400 z/OS

Destructs an iostream_withassign object.

iostream_withassign

```
public:iostream_withassign()
```

This is supported on AIX 400 z/OS

Creates an iostream_withassign object. It does not do any initialization of this object.

operator =

```
public:iostream_withassign& operator =(iostream_withassign& rhs)
```

This is supported on AIX z/OS

Copy constructor.

Assignment Operators

operator =

Overload 1

```
public:iostream_withassign& operator =(streambuf*)
```

This is supported on AIX 400 z/OS

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 is supported on AIX 400 z/OS

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**	**Definition**	**Page Number**
~ios	92	bad	94
bitalloc	99	clear	94
eof	94	fail	94
fill	95	flags	95
good	94	ios	92
ios_resource	98	iword	99
operator !	99	operator const void *	99
operator void *	99	precision	95
pword	99	rdbuf	99
rdstate	94	setf	96
skip	97	sync_with_stdio	100
tie	100	unsetf	97
width	97	xalloc	101

iostream			
Definition	**Page Number**	**Definition**	**Page Number**
~iostream	106	iostream	106

istream			
Definition	**Page Number**	**Definition**	**Page Number**
~istream	111	gcount	112
get	113	get_complicated	116
getline	116	ignore	119
ipfx	133	isfx	134
istream	112	operator >>	122
peek	120	putback	132
read	120	rs_complicated	121
seekg	132	sync	133
tellg	133		

Inherited Public Data

ios			
Definition	**Page Number**	**Definition**	**Page Number**
adjustfield	92	basefield	92
floatfield	92		

Inherited Protected Functions

istream			
Definition	**Page Number**	**Definition**	**Page Number**
do_ipfx	134	eatwhite	122
istream	112		

ios			
Definition	**Page Number**	**Definition**	**Page Number**
init	98	ios	92
setstate	95		

iostream			
Definition	**Page Number**	**Definition**	**Page Number**
iostream	106		

Inherited Protected Data

ios			
Definition	**Page Number**	**Definition**	**Page Number**
(* stdioflush) ( )	101	assign_private	92

ios			
Definition	**Page Number**	**Definition**	**Page Number**
bp	93	delbuf	93
isfx_special	93	ispecial	93
osfx_special	93	ospecial	93
state	93	x_fill	98
x_flags	93	x_precision	98
x_tie	93	x_width	98

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()
```

This is supported on 

Destructs an istream object.

istream

Overload 1

```
public:istream(streambuf*, int sk, ostream* t = 0)
```

This is supported on 

Obsolete. Do not use.

Overload 2

```
public:istream(streambuf*)
```

This is supported on 

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)
```

This is supported on 

Obsolete. Do not use.

Overload 4

```
public:istream(int fd, int sk = 1, ostream* t = 0)
```

This is supported on 

Obsolete. Do not use.

Overload 5

```
protected:istream()
```

This is supported on 

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()
```

This is supported on AIX 400 z/OS

AIX and z/OS Considerations

This function returns an int value for 32-bit applications. It is not available for 64-bit applications.

Overload 2

```
public:long gcount()
```

This is supported on AIX z/OS

This function returns a long value for 64-bit applications. It is not available for 32-bit applications.

get

Overload 1

```
public:int get()
```

This is supported on AIX 400 z/OS

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')
```

This is supported on AIX 400 z/OS

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 and z/OS Considerations

This function is available for 32-bit applications. The second argument is an int value.

Overload 3

```
public:istream& get(unsigned char& c)
```

This is supported on AIX 400 z/OS

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')
```

This is supported on AIX 400 z/OS

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 and z/OS 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')
```

This is supported on AIX 400 z/OS

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')
```

This is supported on AIX z/OS

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.

This function is available for 64-bit applications. The second argument is a long value.

Overload 7

```
public:istream& get(char& c)
```

This is supported on AIX 400 z/OS

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)
```

This is supported on AIX 400 z/OS

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&)
```

This is supported on AIX 400 z/OS

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')
```

This is supported on AIX 400 z/OS

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 and z/OS 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')
```

This is supported on AIX z/OS

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.

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')
```

This is supported on AIX z/OS

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.

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)
```

This is supported on AIX 400 z/OS

Internal function. Do not use.

Overload 2

```
public:istream& get_complicated(unsigned char& c)
```

This is supported on AIX 400 z/OS

Internal function. Do not use.

Overload 3

```
public:istream& get_complicated(char& c)
```

This is supported on AIX 400 z/OS

Internal function. Do not use.

getline

Overload 1

```
public:istream&
  getline( unsigned char* b,
           int lim,
           char delim = '\n' )
```

This is supported on AIX 400 z/OS

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 and z/OS 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' )
```

This is supported on AIX z/OS

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.

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')
```

This is supported on AIX 400 z/OS

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 and z/OS 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')
```

This is supported on AIX z/OS

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.

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' )
```

This is supported on AIX 400 z/OS

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 and z/OS 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' )
```

This is supported on AIX z/OS

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.

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)
```

This is supported on AIX 400 z/OS

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()
```

This is supported on AIX 400 z/OS

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)
```

This is supported on AIX z/OS

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().

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)
```

This is supported on AIX 400 z/OS

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 and z/OS 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)
```

This is supported on AIX z/OS

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().

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)
```

This is supported on AIX 400 z/OS

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 and z/OS 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)
```

This is supported on AIX z/OS

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().

This function is available for 64-bit applications. The second argument is a long value.

Overload 6

```
public:istream& read(char* s, int n)
```

This is supported on AIX 400 z/OS

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 and z/OS 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)
```

This is supported on AIX 400 z/OS

Internal function. Do not use.

Overload 2

```
public:istream& rs_complicated(char& c)
```

This is supported on AIX 400 z/OS

Internal function. Do not use.

Overload 3

```
public:istream& rs_complicated(unsigned char& c)
```

This is supported on AIX 400 z/OS

Internal function. Do not use.

eatwhite

```
protected:void eatwhite()
```

This is supported on AIX 400 z/OS

Internal function. Do not use.

Input Operators

Input operators supported by istream objects.

operator >>

Overload 1

```
public:istream& operator >>(float&)
```

This is supported on AIX 400 z/OS

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*)
```

This is supported on AIX 400 z/OS

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&)
```

This is supported on 

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 from ″A″ to ″F″, upper or lower case) is encountered. If ios::hex is set and a signed value is encountered, the value is converted into a decimal value.

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&)
```

This is supported on 

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 & ))
```

This is supported on AIX 400 z/OS

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&)
```

This is supported on AIX 400 z/OS

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 from ″A″ to ″F″, upper or lower case) is encountered. If ios::hex is set and a signed value is encountered, the value is converted into a decimal value.

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&)
```

This is supported on 

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 from ″A″ to ″F″, upper or lower case) is encountered. If ios::hex is set and a signed value is encountered, the value is converted into a decimal value.

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&)
```

This is supported on AIX 400 z/OS

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 from ″A″ to ″F″, upper or lower case) is encountered. If ios::hex is set and a signed value is encountered, the value is converted into a decimal value.

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)
```

This is supported on AIX 400 z/OS

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*)
```

This is supported on 

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*)
```

This is supported on 

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*)
```

This is supported on 

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&)
```

This is supported on 

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 from ″A″ to ″F″, upper or lower case) is encountered. If ios::hex is set and a signed value is encountered, the value is converted into a decimal value.

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&)
```

This is supported on 

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 from "A" to "F", upper or lower case) is encountered. If ios::hex is set and a signed value is encountered, the value is converted into a decimal value.

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&)
```

This is supported on 

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 from "A" to "F", upper or lower case) is encountered. If ios::hex is set and a signed value is encountered, the value is converted into a decimal value.

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&)
```

This is supported on 

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 from ″A″ to ″F″, upper or lower case) is encountered. If ios::hex is set and a signed value is encountered, the value is converted into a decimal value.

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&)
```

This is supported on AIX 400 z/OS

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*)
```

This is supported on AIX 400 z/OS

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)
```

This is supported on AIX 400 z/OS

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 & ))
```

This is supported on AIX 400 z/OS

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&)
```

This is supported on AIX 400 z/OS

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)
```

This is supported on 

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)
```

This is supported on 

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)
```

This is supported on 

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)
```

This is supported on AIX 400 z/OS

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()
```

This is supported on AIX 400 z/OS

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()
```

This is supported on AIX 400 z/OS

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)
```

This is supported on AIX 400 z/OS

AIX and z/OS 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)
```

This is supported on AIX z/OS

This function accepts a long value for 64-bit applications. It is not available for 32-bit applications.

isfx

```
public:void isfx()
```

This is supported on AIX 400 z/OS

Internal function. Do not use.

do_ipfx

Overload 1

```
protected:int do_ipfx(long noskipws)
```

This is supported on AIX z/OS

Internal function. Do not use.

This function is available for 64-bit applications. It accepts a long argument.

Overload 2

```
protected:int do_ipfx(int noskipws)
```

This is supported on AIX 400 z/OS

Internal function. Do not use.

AIX and z/OS 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**	**Definition**	**Page Number**
~ios	92	bad	94
bitalloc	99	clear	94
eof	94	fail	94
fill	95	flags	95
good	94	ios	92
ios_resource	98	iword	99
operator !	99	operator const void *	99
operator void *	99	precision	95
pword	99	rdbuf	99
rdstate	94	setf	96
skip	97	sync_with_stdio	100
tie	100	unsetf	97
width	97	xalloc	101

Inherited Public Data

ios			
Definition	Page Number	Definition	Page Number
adjustfield floatfield	92 92	basefield	92

Inherited Protected Functions

ios			
Definition	Page Number	Definition	Page Number
init setstate	98 95	ios	92

Inherited Protected Data

ios			
Definition	Page Number	Definition	Page Number
(* stdioflush) ( )	101	assign_private	92
bp	93	delbuf	93
isfx_special	93	ispecial	93
osfx_special	93	ospecial	93
state	93	x_fill	98
x_flags	93	x_precision	98
x_tie	93	x_width	98

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()
```

This is supported on AIX 400 z/OS

Destructs an ostream_withassign object.

istream_withassign

```
public:istream_withassign()
```

This is supported on AIX 400 z/OS

Creates an istream_withassign object. It does not do any initialization of this object.

operator =

```
public:istream_withassign& operator =(istream_withassign& rhs)
```

This is supported on AIX

The copy constructor.

Assignment Operator

Assignment operators for istream_withassign.

operator =

Overload 1

```
public:istream_withassign& operator =(streambuf*)
```

This is supported on AIX 400 z/OS

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 is supported on AIX 400 z/OS

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**	**Definition**	**Page Number**
~ios	92	bad	94
bitalloc	99	clear	94
eof	94	fail	94
fill	95	flags	95
good	94	ios	92
ios_resource	98	iword	99
operator !	99	operator const void *	99
operator void *	99	precision	95
pword	99	rdbuf	99
rdstate	94	setf	96
skip	97	sync_with_stdio	100

ios			
Definition	**Page Number**	**Definition**	**Page Number**
tie	100	unsetf	97
width	97	xalloc	101

istream			
Definition	**Page Number**	**Definition**	**Page Number**
~istream	111	gcount	112
get	113	get_complicated	116
getline	116	ignore	119
ipfx	133	isfx	134
istream	112	operator >>	122
peek	120	putback	132
read	120	rs_complicated	121
seekg	132	sync	133
tellg	133		

Inherited Public Data

ios			
Definition	**Page Number**	**Definition**	**Page Number**
adjustfield	92	basefield	92
floatfield	92		

Inherited Protected Functions

istream			
Definition	**Page Number**	**Definition**	**Page Number**
do_ipfx	134	eatwhite	122
istream	112		

ios			
Definition	**Page Number**	**Definition**	**Page Number**
init	98	ios	92
setstate	95		

Inherited Protected Data

ios			
Definition	**Page Number**	**Definition**	**Page Number**
`( * stdioflush ) ( )`	101	`assign_private`	92
`bp`	93	`delbuf`	93
`isfx_special`	93	`ispecial`	93
`osfx_special`	93	`ospecial`	93
`state`	93	`x_fill`	98
`x_flags`	93	`x_precision`	98
`x_tie`	93	`x_width`	98

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()
```

This is supported on AIX 400 z/OS

The istrstream destructor frees space that was allocated by the istrstream constructor.

istrstream

Overload 1

```
public:istrstream(const char* str)
```

This is supported on AIX 400 z/OS

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 is supported on AIX 400 z/OS

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 is supported on AIX z/OS

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.

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 is supported on AIX z/OS

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.

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 is supported on AIX 400 z/OS

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 and z/OS 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 is supported on AIX z/OS

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.

This function is available for 64-bit applications. The second argument is a long value.

Overload 7

```
public:istrstream(const unsigned char* str)
```

This is supported on AIX 400 z/OS

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 is supported on AIX z/OS

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.

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 is supported on AIX 400 z/OS

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 and z/OS 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 is supported on AIX 400 z/OS

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 and z/OS Considerations

This function is available for 32-bit applications. The second argument is an int value.

Overload 11

```
public:istrstream(signed char* str)
```

This is supported on AIX 400 z/OS

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 is supported on AIX 400 z/OS

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 is supported on AIX 400 z/OS

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 and z/OS 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 is supported on AIX z/OS

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.

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 is supported on AIX 400 z/OS

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 and z/OS 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 is supported on AIX z/OS

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.

This function is available for 64-bit applications. The second argument is a long value.

Overload 17

```
public:istrstream(char* str, int size)
```

This is supported on AIX 400 z/OS

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 and z/OS Considerations

This function is available for 32-bit applications. The second argument is an int value.

Overload 18

```
public:istrstream(char* str)
```

This is supported on AIX 400 z/OS

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**	**Definition**	**Page Number**
~ios	92	bad	94
bitalloc	99	clear	94
eof	94	fail	94
fill	95	flags	95
good	94	ios	92
ios_resource	98	iword	99
operator !	99	operator const void *	99
operator void *	99	precision	95
pword	99	rdbuf	99
rdstate	94	setf	96
skip	97	sync_with_stdio	100
tie	100	unsetf	97
width	97	xalloc	101

strstreambase			
Definition	**Page Number**	**Definition**	**Page Number**
rdbuf	194		

Inherited Public Data

ios			
Definition	**Page Number**	**Definition**	**Page Number**
adjustfield	92	basefield	92
floatfield	92		

Inherited Protected Functions

strstreambase			
Definition	**Page Number**	**Definition**	**Page Number**
~strstreambase	193	strstreambase	193

ios			
Definition	**Page Number**	**Definition**	**Page Number**
init setstate	98 95	ios	92

Inherited Protected Data

ios			
Definition	**Page Number**	**Definition**	**Page Number**
(* stdioflush) ( )	101	assign_private	92
bp	93	delbuf	93
isfx_special	93	ispecial	93
osfx_special	93	ospecial	93
state	93	x_fill	98
x_flags	93	x_precision	98
x_tie	93	x_width	98

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()
```

This is supported on AIX 400 z/OS

Destructs an ofstream object.

ofstream

Constructs an object of this class.

Overload 1

```
public:ofstream(int fd, char* p, int l)
```

This is supported on AIX 400 z/OS

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 and z/OS Considerations

This function is available for 32-bit applications. The third argument is an int value.

Overload 2

```
public:ofstream( const char* name,
          const char* attr,
          int mode = ios::out,
          int prot = filebuf::openprot )
```

This is supported on z/OS

Constructs an ofstream object and opens the file name with open mode equal to mode, attributes equal to attr 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.

You can use the attr parameter to specify additional file attributes such as lrecl or recfm. All the parameters documented for the fopen() functions are supported, with the exception of type=record.

z/OS Considerations

The prot attribute is ignored.

Overload 3

```
public:ofstream(int fd)
```

This is supported on AIX 400 z/OS

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 4

```
public:ofstream(int fd, char* p, long l)
```

This is supported on AIX z/OS

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.

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,
          _CCSID_T ccsid = _CCSID_T ( 0 ) )
```

This is supported on 400

Constructs an ifstream object and opens the file name with open mode equal to mode and protection mode equal to prot, and ccsid equal to ccsid. 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.

If the ccsid parameter is non-zero then it is treated as a CCSID (coded character set identifier) and will correspond to the CCSID of data written to and from the file. If the parameter value is zero then the CCSID of the job will be used.

Overload 6

```
public:ofstream(const char* name, int mode, _CCSID_T)
```

This is supported on 400

Constructs an ofstream object and opens the file name with open mode equal to mode and ccsid equal to ccsid. If the file cannot be opened, the error state of the constructed fstream object is set.

If the ccsid parameter is non-zero then it is treated as a CCSID (coded character set identifier) and will correspond to the CCSID of data written to and from the file. If the parameter value is zero then the CCSID of the job will be used.

Overload 7

```
public:ofstream( const char* name,
          int mode = ios::out,
          int prot = filebuf::openprot )
```

This is supported on AIX z/OS

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.

z/OS Considerations

The prot attribute is ignored.

Overload 8

```
public:ofstream()
```

This is supported on AIX 400 z/OS

Constructs an unopened ofstream object.

Filebuf Functions

rdbuf

```
public:filebuf* rdbuf()
```

This is supported on AIX 400 z/OS

Returns a pointer to the filebuf object that is attached to the ofstream object.

Open Functions

Opens the file.

z/OS Considerations

The prot attribute is ignored.

open

Opens the specified file.

Overload 1

```
public:void
  open( const char* name,
        int mode = ios::out,
        int prot = filebuf::openprot,
        _CCSID_T ccsid = _CCSID_T ( 0 ) )
```

This is supported on 400

Opens the file with the specified name, mode, protection and coded character set id 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 or 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.

If the ccsid parameter is non-zero then it is treated as a CCSID (coded character set identifier) and will correspond to the CCSID of data written to and from the file. If the parameter value is zero then the CCSID of the job will be used.

Overload 2

```
public:void
  open( const char* name,
        int mode = ios::out,
        int prot = filebuf::openprot )
```

This is supported on AIX z/OS

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.

Overload 3

```
public:void open(const char* name, int mode, _CCSID_T ccsid)
```

This is supported on 400

Opens the file with the specified name, mode and coded character set id and attaches it to the fstream object.

If the file with the name, name does not already exist, open() tries to create it unless ios::nocreate is set.

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.

If the ccsid parameter is non-zero then it is treated as a CCSID (coded character set identifier) and will correspond to the CCSID of data written to and from the file. If the parameter value is zero then the CCSID of the job will be used.

Overload 4

```
public:void
  open( const char* name,
        const char* attr,
        int mode = ios::out,
        int prot = filebuf::openprot )
```

This is supported on z/OS

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.

You can use the attr parameter to specify additional file attributes, such as lrecl or recfm. All the parameters documented for the fopen() function are supported, with the exception of type=record.

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**	**Definition**	**Page Number**
~fstreambase	79	attach	81
close	82	detach	82
fstreambase	79	open	82
setbuf	84		

ios			
Definition	**Page Number**	**Definition**	**Page Number**
~ios	92	bad	94
bitalloc	99	clear	94
eof	94	fail	94
fill	95	flags	95
good	94	ios	92
ios_resource	98	iword	99
operator !	99	operator const void *	99
operator void *	99	precision	95
pword	99	rdbuf	99
rdstate	94	setf	96
skip	97	sync_with_stdio	100
tie	100	unsetf	97
width	97	xalloc	101

Inherited Public Data

ios			
Definition	**Page Number**	**Definition**	**Page Number**
adjustfield	92	basefield	92
floatfield	92		

Inherited Protected Functions

ios			
Definition	**Page Number**	**Definition**	**Page Number**
init	98	ios	92
setstate	95		

fstreambase			
Definition	**Page Number**	**Definition**	**Page Number**
verify	82		

Inherited Protected Data

ios			
Definition	**Page Number**	**Definition**	**Page Number**
`( * stdioflush ) ( )`	101	`assign_private`	92
`bp`	93	`delbuf`	93
`isfx_special`	93	`ispecial`	93
`osfx_special`	93	`ospecial`	93
`state`	93	`x_fill`	98
`x_flags`	93	`x_precision`	98
`x_tie`	93	`x_width`	98

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()
```

This is supported on 

Destructs an ostream object.

ostream

Overload 1

```
public:ostream(streambuf*)
```

This is supported on 

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 is supported on 

This constructor is obsolete; do not use it.

Overload 3

```
public:ostream(int size, char*)
```

This is supported on 

This constructor is obsolete; do not use it.

Overload 4

```
protected:ostream()
```

This is supported on 

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)
```

This is supported on 

flush

```
public:ostream& flush()
```

This is supported on 

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.

When ostream::flush() is called, one of the following occurs:

- if the stream buffer's put area is not empty and there are characters waiting to be consumed, flush will call the stream buffer's overflow() function to flush out all the content in the put area.
- if the stream buffer's get area is not empty and there are characters waiting to be extracted, flush will call the stream buffer's sync() function. The sync() function will clean up both the put area and the get area by sending any characters that are stored in the put area to the ultimate consumer, and sending any characters that are waiting in the get area back to the ultimate producer.

ls_complicated

Overload 1

```
public:ostream& ls_complicated(char)
```

This is supported on 

Internal function. Do not use.

Overload 2

```
public:ostream& ls_complicated(signed char)
```

This is supported on 

Internal function. Do not use.

Overload 3

```
public:ostream& ls_complicated(unsigned char)
```

This is supported on 

Internal function. Do not use.

put

```
public:ostream& put(char c)
```

This is supported on 

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)
```

This is supported on 

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)
```

This is supported on 

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)
```

This is supported on 

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*)
```

This is supported on AIX 400 z/OS

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*)
```

This is supported on AIX 400 z/OS

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*)
```

This is supported on AIX 400 z/OS

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 & ))
```

This is supported on AIX 400 z/OS

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)
```

This is supported on 

The output operator inserts the character into the stream buffer without performing any conversion on it.

Overload 6

```
public:ostream& operator <<(unsigned long)
```

This is supported on 

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)
```

This is supported on 

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)
```

This is supported on AIX 400 z/OS

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)
```

This is supported on AIX 400 z/OS

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)
```

This is supported on 

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)
```

This is supported on AIX 400 z/OS

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)
```

This is supported on AIX 400 z/OS

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)
```

This is supported on AIX 400 z/OS

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)
```

This is supported on AIX 400 z/OS

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)
```

This is supported on 

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*)
```

This is supported on 

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)
```

This is supported on AIX 400 z/OS

The output operator inserts the character into the stream buffer without performing any conversion on it.

Overload 18

```
public:ostream& operator <<(float)
```

This is supported on AIX 400 z/OS

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 & ))
```

This is supported on AIX 400 z/OS

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)
```

This is supported on AIX 400 z/OS

The output operator inserts the character into the stream buffer without performing any conversion on it.

Overload 21

```
public:ostream& operator <<(streambuf*)
```

This is supported on AIX 400 z/OS

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*)
```

This is supported on AIX 400 z/OS

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)
```

This is supported on AIX 400 z/OS

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)
```

This is supported on AIX 400 z/OS

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)
```

This is supported on AIX 400 z/OS

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()
```

This is supported on AIX 400 z/OS

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()
```

This is supported on AIX 400 z/OS

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()
```

This is supported on AIX 400 z/OS

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()
```

This is supported on AIX 400 z/OS

Internal function. Do not use.

do_osfx

```
protected:void do_osfx()
```

This is supported on AIX 400 z/OS

Internal function. Do not use.

ostream - Inherited Member Functions and Data

Inherited Public Functions

ios			
Definition	**Page Number**	**Definition**	**Page Number**
~ios	92	bad	94
bitalloc	99	clear	94
eof	94	fail	94
fill	95	flags	95
good	94	ios	92
ios_resource	98	iword	99
operator !	99	operator const void *	99
operator void *	99	precision	95
pword	99	rdbuf	99
rdstate	94	setf	96
skip	97	sync_with_stdio	100
tie	100	unsetf	97
width	97	xalloc	101

Inherited Public Data

ios			
Definition	**Page Number**	**Definition**	**Page Number**
adjustfield	92	basefield	92
floatfield	92		

Inherited Protected Functions

ios			
Definition	**Page Number**	**Definition**	**Page Number**
init	98	ios	92
setstate	95		

Inherited Protected Data

ios			
Definition	**Page Number**	**Definition**	**Page Number**
(* stdioflush) ( )	101	assign_private	92
bp	93	delbuf	93

ios			
Definition	**Page Number**	**Definition**	**Page Number**
isfx_special	93	ispecial	93
osfx_special	93	ospecial	93
state	93	x_fill	98
x_flags	93	x_precision	98
x_tie	93	x_width	98

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()
```

This is supported on AIX 400 z/OS

Destructs an ostream_withassign object.

operator =

```
public:ostream_withassign& operator =(ostream_withassign& rhs)
```

This is supported on AIX

Copy constructor.

ostream_withassign

```
public:ostream_withassign()
```

This is supported on AIX 400 z/OS

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 is supported on AIX 400 z/OS

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 is supported on AIX 400 z/OS

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**	**Definition**	**Page Number**
~ios	92	bad	94
bitalloc	99	clear	94
eof	94	fail	94
fill	95	flags	95
good	94	ios	92
ios_resource	98	iword	99
operator !	99	operator const void *	99
operator void *	99	precision	95
pword	99	rdbuf	99
rdstate	94	setf	96
skip	97	sync_with_stdio	100
tie	100	unsetf	97
width	97	xalloc	101

ostream			
Definition	**Page Number**	**Definition**	**Page Number**
~ostream	149	complicated_put	150
flush	150	ls_complicated	151
operator <<	152	opfx	161
osfx	161	ostream	150
put	151	seekp	160
tellp	161	write	151

Inherited Public Data

ios			
Definition	**Page Number**	**Definition**	**Page Number**
adjustfield floatfield	92 92	basefield	92

Inherited Protected Functions

ostream			
Definition	**Page Number**	**Definition**	**Page Number**
do_opfx ostream	161 150	do_osfx	162

ios			
Definition	**Page Number**	**Definition**	**Page Number**
init setstate	98 95	ios	92

Inherited Protected Data

ios			
Definition	**Page Number**	**Definition**	**Page Number**
(* stdioflush) ( )	101	assign_private	92
bp	93	delbuf	93
isfx_special	93	ispecial	93
osfx_special	93	ospecial	93
state	93	x_fill	98
x_flags	93	x_precision	98
x_tie	93	x_width	98

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 - Member Functions and Data by Group

Constructors & Destructor

Objects of the ostrstream class can be constructed and destructed.

~ostrstream

```
public:~ostrstream()
```

This is supported on AIX 400 z/OS

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 is supported on AIX 400 z/OS

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 and z/OS 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 is supported on AIX z/OS

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.

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 is supported on AIX z/OS

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.

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 is supported on AIX z/OS

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.

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 is supported on AIX 400 z/OS

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 and z/OS 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 is supported on AIX 400 z/OS

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 and z/OS Considerations

This function is available for 32-bit applications. The second argument is an int value.

Overload 7

```
public:ostrstream()
```

This is supported on AIX 400 z/OS

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()
```

This is supported on AIX 400 z/OS

AIX and z/OS Considerations

This function returns an int value for 32-bit applications. It is not available for 64-bit applications.

Overload 2

```
public:long pcount()
```

This is supported on AIX z/OS

This function returns a long value for 64-bit applications. It is not available for 32-bit applications.

str

```
public:char* str()
```

This is supported on AIX 400 z/OS

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**	**Definition**	**Page Number**
`~ios`	92	`bad`	94
`bitalloc`	99	`clear`	94
`eof`	94	`fail`	94

ios			
Definition	**Page Number**	**Definition**	**Page Number**
fill	95	flags	95
good	94	ios	92
ios_resource	98	iword	99
operator !	99	operator const void *	99
operator void *	99	precision	95
pword	99	rdbuf	99
rdstate	94	setf	96
skip	97	sync_with_stdio	100
tie	100	unsetf	97
width	97	xalloc	101

strstreambase			
Definition	**Page Number**	**Definition**	**Page Number**
rdbuf	194		

Inherited Public Data

ios			
Definition	**Page Number**	**Definition**	**Page Number**
adjustfield	92	basefield	92
floatfield	92		

Inherited Protected Functions

strstreambase			
Definition	**Page Number**	**Definition**	**Page Number**
~strstreambase	193	strstreambase	193

ios			
Definition	**Page Number**	**Definition**	**Page Number**
init	98	ios	92
setstate	95		

Inherited Protected Data

ios			
Definition	**Page Number**	**Definition**	**Page Number**
(* stdioflush) ( )	101	assign_private	92

ios			
Definition	**Page Number**	**Definition**	**Page Number**
bp	93	delbuf	93
isfx_special	93	ispecial	93
osfx_special	93	ospecial	93
state	93	x_fill	98
x_flags	93	x_precision	98
x_tie	93	x_width	98

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()
```

This is supported on AIX 400 z/OS

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)
```

This is supported on AIX 400 z/OS

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)
```

This is supported on AIX 400 z/OS

Emptys an output buffer. Returns EOF on error, 0 otherwise.

pbackfail

```
public:virtual int pbackfail(int c)
```

This is supported on AIX 400 z/OS

Attempts to put back a character.

seekoff

```
public:virtual streampos seekoff(streamoff, ios::seek_dir, int)
```

This is supported on AIX 400 z/OS

sync

```
public:virtual int sync()
```

This is supported on AIX 400 z/OS

underflow

```
public:virtual int underflow()
```

This is supported on AIX 400 z/OS

Fills an input buffer. Returns EOF on error or end of input, 0 otherwise.

Query Functions

stdiofile

```
public:FILE* stdiofile()
```

This is supported on AIX 400 z/OS

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**	**Definition**	**Page Number**
~streambuf	175	dbp	178
in_avail	176	optim_in_avail	176
optim_sbumpc	176	out_waiting	183
overflow	183	pptr_non_null	179
sbumpc	176	seekoff	179
seekpos	179	setbuf	185
sgetc	176	sgetn	177
snextc	177	sputbackc	184
sputc	184	sputn	184
stossc	180	streambuf	175
streambuf_resource	185	xsgetn	178
xsputn	185		

Inherited Public Data

None

Inherited Protected Functions

streambuf			
Definition	**Page Number**	**Definition**	**Page Number**
allocate	187	base	180
blen	187	doallocate	188
eback	180	ebuf	180
egptr	180	epptr	180
gbump	181	gptr	181
pbase	181	pbump	181
pptr	182	setb	182
setg	182	setp	182
unbuffered	188		

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()
```

This is supported on AIX 400 z/OS

Destructs a stdiostream object.

stdiostream

```
public:stdiostream(FILE*)
```

This is supported on AIX 400 z/OS

Creates a stdiostream object that is attached to the FILE pointed to by the argument.

Miscellaneous

rdbuf

```
public:stdiobuf* rdbuf()
```

This is supported on AIX 400 z/OS

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**	**Definition**	**Page Number**
~ios	92	bad	94
bitalloc	99	clear	94
eof	94	fail	94
fill	95	flags	95
good	94	ios	92
ios_resource	98	iword	99
operator !	99	operator const void *	99
operator void *	99	precision	95
pword	99	rdbuf	99
rdstate	94	setf	96
skip	97	sync_with_stdio	100
tie	100	unsetf	97
width	97	xalloc	101

Inherited Public Data

ios			
Definition	**Page Number**	**Definition**	**Page Number**
adjustfield	92	basefield	92
floatfield	92		

Inherited Protected Functions

ios			
Definition	**Page Number**	**Definition**	**Page Number**
init	98	ios	92
setstate	95		

Inherited Protected Data

ios			
Definition	**Page Number**	**Definition**	**Page Number**
(* stdioflush) ( )	101	assign_private	92
bp	93	delbuf	93
isfx_special	93	ispecial	93
osfx_special	93	ospecial	93
state	93	x_fill	98
x_flags	93	x_precision	98
x_tie	93	x_width	98

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()
```

This is supported on AIX 400 z/OS

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)
```

This is supported on AIX z/OS

Constructs an empty stream buffer of length l starting at the position pointed to by p.

This constructor is available for 64-bit applications. The second argument is a long value.

Overload 2

```
public:streambuf(char* p, int l)
```

This is supported on AIX 400 z/OS

Constructs an empty stream buffer of length l starting at the position pointed to by p.

AIX and z/OS 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 is supported on AIX 400 z/OS

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()
```

This is supported on AIX 400 z/OS

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()
```

This is supported on AIX 400 z/OS

AIX and z/OS 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()
```

This is supported on AIX z/OS

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()
```

This is supported on AIX 400 z/OS

Returns true if the current get pointer is less then the end of the get area.

optim_sbumpc

```
public:int optim_sbumpc()
```

This is supported on AIX 400 z/OS

Moves the get pointer past one character and returns the character that it moved past.

sbumpc

```
public:int sbumpc()
```

This is supported on AIX 400 z/OS

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()
```

This is supported on AIX 400 z/OS

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)
```

This is supported on AIX z/OS

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.

This function is available for 64-bit applications. It accepts a long argument.

Overload 2

```
public:int sgetn(char* s, int n)
```

This is supported on AIX 400 z/OS

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 and z/OS Considerations

This function is available for 32-bit applications. It accepts an int argument.

snextc

```
public:int snextc()
```

This is supported on AIX 400 z/OS

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()
```

This is supported on AIX 400 z/OS

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)
```

This is supported on AIX 400 z/OS

Similar to sputn.

AIX and z/OS 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)
```

This is supported on AIX z/OS

Similar to sgetn.

This function is available for 64-bit applications. The second argument is a long value.

Get/Put Pointer Functions

dbp

```
public:void dbp()
```

This is supported on AIX 400 z/OS

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()
```

This is supported on AIX

Returns true if the put pointer is not null.

seekoff

```
public:virtual streampos
  seekoff( streamoff,
           ios::seek_dir,
           int = ios::in|ios::out )
```

This is supported on AIX 400 z/OS

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 )
```

This is supported on AIX 400 z/OS

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()
```

This is supported on 

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()
```

This is supported on 

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()
```

This is supported on 

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()
```

This is supported on 

Returns a pointer to the byte after the last byte of the stream buffer.

egptr

```
protected:char* egptr()
```

This is supported on 

Returns a pointer to the byte after the last byte of the get area of the streambuf.

epptr

```
protected:char* epptr()
```

This is supported on 

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)
```

This is supported on AIX z/OS

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().

This function is available for 64-bit applications. It accepts a long argument.

Overload 2

```
protected:void gbump(int n)
```

This is supported on AIX 400 z/OS

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 and z/OS Considerations

This function is available for 32-bit applications. It accepts an int argument.

gptr

```
protected:char* gptr()
```

This is supported on AIX 400 z/OS

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()
```

This is supported on AIX 400 z/OS

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)
```

This is supported on AIX z/OS

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().

This function is available for 64-bit applications. It accepts a long argument.

Overload 2

```
protected:void pbump(int n)
```

This is supported on AIX 400 z/OS

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 and z/OS Considerations

This function is available for 32-bit applications. It accepts an int argument.

pptr

```
protected:char* pptr()
```

This is supported on AIX 400 z/OS

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)
```

This is supported on AIX 400 z/OS

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)
```

This is supported on AIX 400 z/OS

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)
```

This is supported on AIX 400 z/OS

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()
```

This is supported on AIX 400 z/OS

AIX and z/OS 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()
```

This is supported on AIX z/OS

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)
```

This is supported on AIX 400 z/OS

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)
```

This is supported on AIX 400 z/OS

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)
```

This is supported on AIX 400 z/OS

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)
```

This is supported on AIX 400 z/OS

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)
```

This is supported on AIX 400 z/OS

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 and z/OS Considerations

This function is available for 32-bit applications. It accepts an int argument.

Overload 2

```
public:long sputn(const char* s, long n)
```

This is supported on AIX z/OS

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.

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)
```

This is supported on AIX 400 z/OS

Similar to sputn.

AIX and z/OS 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)
```

This is supported on AIX z/OS

Similar to sputn.

This function is available for use when building 64-bit applications. The second argument is a long value.

Locking functions

streambuf_resource

```
public:IRTLResource& streambuf_resource() const
```

This is supported on z/OS

Stream Buffer Functions

Functions that work with the underlying streambuf object.

setbuf

Overload 1

```
public:streambuf* setbuf(unsigned char* p, long len)
```

This is supported on AIX z/OS

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.

This function is available for 64-bit applications. It accepts an long argument.

Overload 2

```
public:virtual streambuf* setbuf(char* p, long len)
```

This is supported on AIX z/OS

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.

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 is supported on AIX 400 z/OS

This function is obsolete. The I/O Stream Library includes it to be compatible with AT&T C++ Language System Release 1.2

AIX and z/OS Considerations

This function is available for 32-bit applications. It accepts an int argument.

Overload 4

```
public:virtual streambuf* setbuf(char* p, int len)
```

This is supported on AIX 400 z/OS

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 and z/OS Considerations

This function is available for 32-bit applications. It accepts an int argument.

Overload 5

```
public:streambuf* setbuf(unsigned char* p, int len)
```

This is supported on AIX 400 z/OS

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 and z/OS Considerations

This function is available for 32-bit applications. It accepts an int argument.

sync

```
public:virtual int sync()
```

This is supported on AIX 400 z/OS

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()
```

This is supported on AIX 400 z/OS

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
```

This is supported on AIX z/OS

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
```

This is supported on AIX 400 z/OS

AIX and z/OS 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()
```

This is supported on AIX 400 z/OS

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)
```

This is supported on AIX 400 z/OS

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
```

This is supported on AIX 400 z/OS

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()
```

This is supported on AIX 400 z/OS

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

strstream

Overload 1

```
public:strstream(char* str, long size, int mode)
```

This is supported on AIX z/OS

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.

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 is supported on AIX 400 z/OS

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 and z/OS 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 is supported on AIX z/OS

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.

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 is supported on AIX 400 z/OS

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 and z/OS 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 is supported on AIX 400 z/OS

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 and z/OS 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 is supported on AIX z/OS

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.

This function is available for 64-bit applications. The second argument is a long value.

Overload 7

```
public:strstream()
```

This is supported on AIX 400 z/OS

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()
```

This is supported on AIX 400 z/OS

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**	**Definition**	**Page Number**
~ios	92	bad	94
bitalloc	99	clear	94
eof	94	fail	94
fill	95	flags	95
good	94	ios	92
ios_resource	98	iword	99
operator !	99	operator const void *	99
operator void *	99	precision	95
pword	99	rdbuf	99
rdstate	94	setf	96
skip	97	sync_with_stdio	100
tie	100	unsetf	97
width	97	xalloc	101

strstreambase			
Definition	**Page Number**	**Definition**	**Page Number**
rdbuf	194		

Inherited Public Data

ios			
Definition	**Page Number**	**Definition**	**Page Number**
adjustfield	92	basefield	92
floatfield	92		

Inherited Protected Functions

strstreambase			
Definition	**Page Number**	**Definition**	**Page Number**
~strstreambase	193	strstreambase	193

ios			
Definition	**Page Number**	**Definition**	**Page Number**
init	98	ios	92
setstate	95		

Inherited Protected Data

ios			
Definition	**Page Number**	**Definition**	**Page Number**
`( * stdioflush ) ( )`	101	`assign_private`	92
`bp`	93	`delbuf`	93
`isfx_special`	93	`ispecial`	93
`osfx_special`	93	`ospecial`	93
`state`	93	`x_fill`	98
`x_flags`	93	`x_precision`	98
`x_tie`	93	`x_width`	98

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

istrstream

ostrstream

strstream

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()
```

This is supported on AIX 400 z/OS

Destructs a strstreambase object.

strstreambase

Overload 1

```
protected:strstreambase(char*, long, char*)
```

This is supported on AIX z/OS

Constructs a strstreambase object.

This function is available for 64-bit applications. The second argument is a long value.

Overload 2

```
protected:strstreambase(char*, int, char*)
```

This is supported on AIX 400 z/OS

Constructs a strstreambase object.

AIX and z/OS Considerations

This function is available for 32-bit applications. The second argument is an int value.

Overload 3

```
protected:strstreambase()
```

This is supported on AIX 400 z/OS

Constructs a strstreambase object.

Misc

rdbuf

```
public:strstreambuf* rdbuf()
```

This is supported on AIX 400 z/OS

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**	**Definition**	**Page Number**
~ios	92	bad	94
bitalloc	99	clear	94
eof	94	fail	94
fill	95	flags	95
good	94	ios	92
ios_resource	98	iword	99
operator !	99	operator const void *	99
operator void *	99	precision	95
pword	99	rdbuf	99
rdstate	94	setf	96
skip	97	sync_with_stdio	100
tie	100	unsetf	97
width	97	xalloc	101

Inherited Public Data

ios			
Definition	**Page Number**	**Definition**	**Page Number**
adjustfield	92	basefield	92
floatfield	92		

Inherited Protected Functions

ios			
Definition	**Page Number**	**Definition**	**Page Number**
`init`	98	`ios`	92
`setstate`	95		

Inherited Protected Data

ios			
Definition	**Page Number**	**Definition**	**Page Number**
`( * stdioflush ) ( )`	101	`assign_private`	92
`bp`	93	`delbuf`	93
`isfx_special`	93	`ispecial`	93
`osfx_special`	93	`ospecial`	93
`state`	93	`x_fill`	98
`x_flags`	93	`x_precision`	98
`x_tie`	93	`x_width`	98

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()
```

This is supported on AIX 400 z/OS

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 is supported on AIX z/OS

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.

This constructor is available for 64-bit applications. It accepts a long argument.

Overload 2

```
public:strstreambuf(int)
```

This is supported on AIX 400 z/OS

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 and z/OS 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)
```

This is supported on AIX 400 z/OS

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 and z/OS 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 )
```

This is supported on AIX 400 z/OS

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 and z/OS 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 )
```

This is supported on AIX z/OS

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.

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 is supported on AIX 400 z/OS

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 )
```

This is supported on AIX 400 z/OS

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 and z/OS 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 )
```

This is supported on AIX z/OS

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.

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)
```

This is supported on AIX z/OS

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.

This constructor is available for 64-bit applications. The second argument is a long value.

Overload 10

```
public:strstreambuf()
```

This is supported on 

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)
```

This is supported on 

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)
```

This is supported on AIX 400 z/OS

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()
```

This is supported on AIX z/OS

This function returns a long for 64-bit applications. It is not available for 32-bit applications.

Overload 2

```
public:int pcount()
```

This is supported on AIX 400 z/OS

AIX and z/OS Considerations

This function returns an int for 32-bit applications. It is not available for 64-bit applications.

underflow

```
public:virtual int underflow()
```

This is supported on AIX 400 z/OS

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()
```

This is supported on AIX 400 z/OS

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)
```

This is supported on AIX 400 z/OS

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()
```

This is supported on AIX 400 z/OS

Returns true if the stream buffer is frozen.

setbuf

Overload 1

```
public:virtual streambuf* setbuf(char* p, long l)
```

This is supported on AIX z/OS

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.

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)
```

This is supported on AIX 400 z/OS

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 and z/OS Considerations

This function is available for 32-bit applications. The second argument is an int value.

str

```
public:char* str()
```

This is supported on AIX 400 z/OS

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**	**Definition**	**Page Number**
~streambuf	175	dbp	178
in_avail	176	optim_in_avail	176
optim_sbumpc	176	out_waiting	183
overflow	183	pbackfail	183
pptr_non_null	179	sbumpc	176
seekoff	179	seekpos	179
setbuf	185	sgetc	176
sgetn	177	snextc	177
sputbackc	184	sputc	184
sputn	184	stossc	180
streambuf	175	streambuf_resource	185
sync	187	xsgetn	178
xsputn	185		

Inherited Public Data

None

Inherited Protected Functions

streambuf			
Definition	**Page Number**	**Definition**	**Page Number**
allocate	187	base	180
blen	187	eback	180
ebuf	180	egptr	180
epptr	180	gbump	181
gptr	181	pbase	181
pbump	181	pptr	182
setb	182	setg	182
setp	182	unbuffered	188

Inherited Protected Data

None

Notices

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.

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. Laboratory
B3/KB7/8200/MKM
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.

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.

Note: 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
- IBM
- OS/390
- OS/400
- VisualAge
- z/OS

UNIX is a registered trademarks of The Open Group in the United States and other countries.

Other company, product, and service names may be trademarks or service marks of others.

SC09-7652-02