

VisualAge<sup>®</sup> C++ Professional for AIX<sup>®</sup>



# IBM<sup>®</sup> Open Class<sup>™</sup> : Object-Persistent Streaming

*Version 5.0*

**Note!**

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

**Edition Notice**

This edition applies to Version 5.0 of IBM VisualAge C++ and to all subsequent releases and modifications until otherwise indicated in new editions. Make sure you are using the correct edition for the level of the product.

© Copyright International Business Machines Corporation 1998, 2000. All rights reserved.

US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

---

# Contents

<b>Notices</b> . . . . .	<b>v</b>
Programming Interface Information . . . . .	vii
Trademarks and Service Marks . . . . .	vii
Industry Standards . . . . .	viii

<b>About This Book.</b> . . . . .	<b>ix</b>
-----------------------------------	-----------

## Chapter 1. IBM Open Class Streaming

### Classes . . . . . **1**

Object Streaming . . . . .	2
Data Streams . . . . .	3
Instantiate a Data Stream . . . . .	3
Instantiate a Stream Module . . . . .	4
Stream Data . . . . .	5
Stream Base Classes . . . . .	6
Application Data Interfaces . . . . .	7
Add Streaming Support to Structs and Simple Classes . . . . .	7
Create a Streamable Class . . . . .	8
Create a Streamable Template Class . . . . .	10
Release-to-Release Data Compatibility (RRDC). . . . .	11
Enable Release-to-Release Data Compatibility . . . . .	12
Exceptions Defined by the Streaming Classes . . . . .	13
Migrate from Older Streaming Constructs . . . . .	14

### Chapter 2. I/O Stream Classes. . . . . **17**

The I/O Stream Class Hierarchy . . . . .	18
I/O Stream Header Files . . . . .	19

The I/O Stream Classes and <code>stdio.h</code> . . . . .	20
Use Predefined Streams . . . . .	20
Stream Buffers . . . . .	21
Format State Flags . . . . .	23
Format Stream Output. . . . .	23
Define Your Own Format State Flags . . . . .	28
Manipulators . . . . .	31
Create Manipulators . . . . .	32
Define an APP Parameterized Manipulator . . . . .	34
Define a MANIP Parameterized Manipulator . . . . .	34
Define Nonassociative Parameterized Manipulators . . . . .	35
Basic I/O Stream Tasks . . . . .	36
Receive Input from Standard Input . . . . .	36
Display Output on Standard Output or Standard Error . . . . .	38
Flush Output Streams with <code>endl</code> and <code>flush</code> . . . . .	40
Parse Multiple Inputs . . . . .	42
Open and Read from Files . . . . .	43
Open and Write to Files . . . . .	46
Combine Input and Output of Different Types. . . . .	46
Advanced I/O Stream Tasks. . . . .	47
Associate a File with a Standard Input or Output Stream . . . . .	47
Move through a file with <code>filebuf</code> Functions . . . . .	48
Define an Input Operator for a Class Type . . . . .	51
Define an Output Operator for a Class Type . . . . .	53
Correct Input Stream Errors . . . . .	55
Manipulate Strings with the <code>stringstream</code> Classes. . . . .	58



---

## Notices

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

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

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

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

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

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

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

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

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

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.  
1150 Eglinton Avenue East  
Toronto, Ontario M3C 1H7  
Canada

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

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

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

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

#### COPYRIGHT LICENSE:

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

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

® (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. ® Copyright IBM Corp. 1998, 2000. All rights reserved.

---

## Programming Interface Information

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

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

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

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

---

## Trademarks and Service Marks

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

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

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

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

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

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

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

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

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

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

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

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

---

## Industry Standards

VisualAge C++ Professional for AIX, Version 5.0 supports the following standards:

- The C language is consistent with the International Standard C (ANSI/ISO-IEC 9899-1990 [1992]). This standard has officially replaced American National standard for Information Systems-Programming Language C (X3.159-1989) and is technically equivalent to the ANSI C standard. VisualAge C++ supports the changes adopted into the C Standard by ISO/IEC 9899:1990/Amendment 1:1994.
- The IBM Systems Application Architecture (SAA) C Level 2 language definition.
- The C++ language is consistent with the International Standard for Information Systems-Programming Language C++ (ISO/IEC 14882:1998).
- The ISO/IEC 9945-1:1990/IEEE POSIX 1003.-1990 standard.
- The X/Open Common Applications Environment Specifications, System Interfaces and Headers, Issue 4.



---

## About This Book

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

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

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



---

## Chapter 1. IBM Open Class Streaming Classes

The IBM Open Class Streaming Classes provide support for the storage and retrieval of data, including both C++ objects and primitive types, from streams. The Streaming Classes comprise three core elements:

- Data streams
- The IMStreamable mixin class
- Stream modules

The Streaming Classes also define several exception classes. The topic “Exceptions Defined by the Streaming Classes” on page 13 lists these exception classes.

Capabilities supported by the Streaming Classes include monomorphic streaming, polymorphic streaming, platform-to-platform data compatibility (a document streamed out on one platform will be readable on another platform), and release-to-release data compatibility (a document written by one version of an application or system will be readable by a newer or an older version).

Typical usage of the Streaming Classes involves the following:

- Implementing persistent storage
- Marshalling function arguments for remote procedure-calls
- Streaming data polymorphically and monomorphically
- Implementing release-to-release data compatibility
- Streaming aliases and aliased objects

### RELATED CONCEPTS

“Object Streaming” on page 2

“Release-to-Release Data Compatibility (RRDC)” on page 11

“Data Streams” on page 3

“Application Data Interfaces” on page 7

“Exceptions Defined by the Streaming Classes” on page 13

Exceptions in the IBM Open Class

### RELATED TASKS

“Add Streaming Support to Structs and Simple Classes” on page 7

“Create a Streamable Class” on page 8

“Create a Streamable Template Class” on page 10

“Instantiate a Stream Module” on page 4

“Enable Release-to-Release Data Compatibility” on page 12

“Stream Base Classes” on page 6

“Instantiate a Data Stream” on page 3

“Stream Data” on page 5

“Migrate from Older Streaming Constructs” on page 14

---

## Object Streaming

### Aliased Objects

Streaming supports the detection and correct handling of aliased pointers to objects.

Suppose that:

- A and B are two objects to stream out
- A and B each have a pointer to the same third object C
- Both A and B are designed to write C to the stream
- When A and B are streamed back in, you need to create one object C to which both A and B point

Without the detection of aliased objects, writing A and B to the stream would cause two copies of C to be written to the stream. Reading the streamed data back in would then result in the creation of two copies of C, one associated with A, the other with B. This would not only waste memory, but would also be logically incorrect. Changes made in one copy of C would not be reflected in the other, and thus would not be seen by both A and B.

With the appropriate use of `writeAliasedObject` instead of plain `writeObject`, the duplicate write of C would be recognized, and would be replaced by a flag, on the stream, that refers back to the initial write. When A and B are read, only one C would be created. The pointers in both A and B would refer to this single object.

Note that for some derived classes with multiple virtual inheritance, base classes may be reachable through multiple paths of the inheritance hierarchy. When a client streams a single instance of the derived class, the base class streaming functions may be invoked more than once. However, streaming classes ensure that there will no adverse effect on the results.

### Monomorphic Streaming

Monomorphic streaming is used when you know the exact type of the data you are streaming. Compared to polymorphic streaming, this can save some overhead. Monomorphic streaming is useful for simple data types, and structs and objects without vtables.

### Polymorphic Streaming

Polymorphic streaming is used to stream an object given a pointer or reference to one of its base classes. This is similar to calling a function which takes a base class pointer, but passing in a pointer to a derived type. When a client reads from the stream, objects of the correct type are automatically constructed and initialized from the stream data.

Consider, for example, the subclasses of `IMGraphic`. A typical drawing has lines, ellipses, and curves, as well as polygons. If you are streaming out many of these objects, you don't want to have to remember the precise sequence of data types that you have streamed out. Polymorphism lets you classify all these objects as an `IMGraphic`. You can then let the runtime type system figure out the precise class of the object being streamed in to a pointer of type `IMGraphic`.

Polymorphic streaming, however, incurs more overhead than monomorphic streaming in terms of time and space. Polymorphic streaming includes the type of each object along with the object's state in the external (stream) representation of

the object. This extra information gives the runtime system the information it needs to reconstruct the correct object when it is streamed back in.

**RELATED CONCEPTS**

“Chapter 1. IBM Open Class Streaming Classes” on page 1

**RELATED TASKS**

“Stream Data” on page 5 “Migrate from Older Streaming Constructs” on page 14

---

## Data Streams

You get an `IDataStream` object when you create a stream. This object handles the encoding of data onto the stream, the decoding of data off of the stream, and the buffering and physical IO of the stream data. `IDataStream` is not intended to be subclassed, instantiated directly, or copied. Data streams are created using stream creation functions from other classes (for example, `IFile::createStream`).

You can write objects to streams in any order you choose, so long as you read corresponding objects in the same order.

You must know the type of an object that has been written to a stream in order to read it from the stream. When you stream in an object pointer, you must stream into a pointer that is the same type as the object on the stream or one of the object’s base classes.

If you write an object monomorphically, you must read it monomorphically. If you write an object polymorphically, you must read it polymorphically. Monomorphic streaming writes out only the bits that represent the value of the streamed object, while polymorphic streaming writes out object information in addition to the object’s value.

**RELATED CONCEPTS**

“Chapter 1. IBM Open Class Streaming Classes” on page 1

**RELATED TASKS**

“Stream Data” on page 5  
“Instantiate a Data Stream”

### Instantiate a Data Stream

The constructors for the `IDataStream` class are private and intended only for internal use. You have two alternatives:

- If you want to read from or write to a file, you can instantiate an `IDataStream` if you have an `IFile` object. See the topic [Access the Contents of Files](#) for more information.
- If you want to stream data into or out of memory, you can instantiate a memory stream, using the static `IDataStream::createMemoryStream` function.

You will typically use a memory stream in a test program to verify the operation of the streaming functions of other classes. You create the stream, write to it, reset it and then read from it. The backing memory grows as necessary to hold the data. When you instantiate a memory stream, you can supply an optional stream format encoding. The default is `IDataStream::kInteroperableBinary`.

- `IDataStream::kRawBinary` — This is a host-specific binary format. It has the best performance, no cross-platform interoperability, and no support for RRDC.

- `IDataStream::kInteroperableBinary` — This is the canonical binary format. It supports platform-to-platform and release-to-release data compatibility.
- `IDataStream::kDebug` — In this mode, the stream contents are printable ASCII and annotated with comments to aid in their interpretation.

To instantiate a memory stream with the default encoding, `kInteroperableBinary`, use the following code:

```
IDataStream* memoryStream = IDataStream::createMemoryStream();
```

To use specify a stream format encoding, pass the appropriate enumerated value to the `createMemoryStream` function:

```
IDataStream* memoryStream = IDataStream::createMemoryStream(IDataStream::kRawBinary);
IDataStream* memoryStream = IDataStream::createMemoryStream(IDataStream::kInteroperableBinary);
IDataStream* memoryStream = IDataStream::createMemoryStream(IDataStream::kDebug);
```

#### RELATED CONCEPTS

“Chapter 1. IBM Open Class Streaming Classes” on page 1

“Data Streams” on page 3

#### RELATED TASKS

“Add Streaming Support to Structs and Simple Classes” on page 7

“Stream Data” on page 5

“Stream Base Classes” on page 6

Access the Contents of Files

## Instantiate a Stream Module

Typically, you will instantiate one instance of `IStreamModule` per library or executable. Instances of `IStreamModule` are used to uniquely identify types on a stream.

You must declare all stream modules as global variables. Do not declare them static because they will not be visible to functions in other files.

```
IStreamModule gMyDataStreamModule("MyDataStreamModule");
```

In the source file for each of your streamable classes, make sure to associate your class with the stream module. The second parameter of the `StreamableDeclarationsMacro` expects the name of the module variable, not the module name string:

```
// In source file for IChangeLanguageCommand
extern IStreamModule gMyDataStreamModule;
StreamableDefinitionsMacro(IChangeLanguageCommand, gMyDataStreamModule);
```

You can also declare the `IStreamModule` in one of your header files so that you don’t have to declare the `IStreamModule` as “extern” in every implementation file that uses it.

#### RELATED CONCEPTS

“Chapter 1. IBM Open Class Streaming Classes” on page 1

“Data Streams” on page 3

#### RELATED TASKS

“Create a Streamable Class” on page 8

“Create a Streamable Template Class” on page 10

## Stream Data

### Stream Aliases and Aliased Objects

It is critical that the objects not be deleted while the stream is still holding references to them, and this is exactly the functionality that `IMRefCounted` provides.

Prior to streaming data, a context frame should be enabled. If your function is streaming a series of objects that contain aliases, and it is important that those aliases be preserved, either your function or some streaming function further up the call chain must establish the context frame. For example, your implementation of the `writeToStream` function for a collection element class does not need to create a context explicitly, because a `writeObject` call upon the collection class holding the element will result in a call to your `writeToStream` function, and a context frame will already exist. But if some other function is iterating through the collection and writing objects to a stream, outside the context of a `writeObject` call, then that iterating function must create a context frame for the stream data.

Consider this example:

```
aStream = IDataStream::createMemoryStream();
{
    IStreamContextFrame aContext(*aStream);
    aStream->writeObject(obj1);
    aStream->writeObject(obj2);
}
// The context is ended when the IStreamContextFrame object goes out
// of scope.
delete aStream;
```

An additional object that is referenced by both `obj1` and `obj2` will appear on the stream only once. After an object has been written to the stream once, any additional writes of the same object will be recognized as duplicates, and a tag referring to the first instance of the object will be written instead.

Each top level `writeObject` and `writeAliasedObject` function automatically creates a context that lasts for the duration of that `writeObject` or `writeAliasedObject` function call. If one of the objects being streamed (`obj1`, for example) contains multiple references to an additional object, only one instance of that additional object will be streamed as consequence of streaming the enclosing object (`obj1`). In this case, you need not explicitly create a context.

### Stream Objects Monomorphically

To stream objects out monomorphically, create a stream and use the stream out operator (`operator>>=`). In the following example, the call to `IDataStream::createMemoryStream()` creates the data stream. The stream out operator writes primitive data to the stream. This overloaded operator also writes out the object `mySimpleObject` according to the implementation of `mySimpleObject::writeToStream`.

```
IDataStream* myStream = IDataStream::createMemoryStream();
myData >>= *myStream;
mySimpleObject >>= *myStream;
```

To stream objects back in monomorphically, use the stream in operator (`operator<<=`):

```
myData <<= *myStream;
mySimpleObject <<= *myStream;
```

### Stream Objects Polymorphically

The stream out function `::writeObject` polymorphically writes the object aliased by `objectPtr` to the stream. Given an `IDataStream*`:

```
Base* objectPtr = new Derived;
::writeObject(objectPtr, *myStream);
```

To stream the object back in polymorphically, use the polymorphic stream in function `::readObject`. Given an `IDataStream&`:

```
Base* objectPtr;
::readObject(objectPtr, myStream);
```

This will “resurrect” an instance of the class `Derived` and assign `objectPtr` a pointer to the derived class instance.

#### RELATED CONCEPTS

“Chapter 1. IBM Open Class Streaming Classes” on page 1  
“Object Streaming” on page 2

#### RELATED TASKS

“Add Streaming Support to Structs and Simple Classes” on page 7  
“Instantiate a Data Stream” on page 3  
“Stream Base Classes”

## Stream Base Classes

The streaming functions for a derived class must explicitly invoke the streaming functions for each base class, in order to stream the full state of the object. Given a class `Derived` which inherits from class `Base`:

```
void Derived::writeToStream(IDataStream& toWhere) const
{
    IStreamOutFrame localFrame(toWhere);
    Base::writeToStream(toWhere); // Stream out our base class
    fData >>= toWhere;           // then stream out our data members
}
```

The `readFromStream` function has the same general form:

```
void Derived::readFromStream(IDataStream& fromWhere)
{
    IStreamInFrame localFrame(fromWhere);
    Base::readFromStream(); // Stream in our base class
    fData <<= fromWhere; // then stream in our data members.
};
```

#### RELATED CONCEPTS

“Chapter 1. IBM Open Class Streaming Classes” on page 1  
“Data Streams” on page 3

#### RELATED TASKS

“Add Streaming Support to Structs and Simple Classes” on page 7  
“Instantiate a Data Stream” on page 3  
“Stream Data” on page 5



---

## Application Data Interfaces

A class that supports streaming should provide, as part of its documentation, a description of its data format on a stream. This application data interface (ADI) will enable developers to create, without access to the original source code, programs that can read the streamed data.

The ADI should include the type (class or primitive) and purpose of each item written by the stream out function of the class, in the order that they are written. In the case of polymorphic streaming, the ADI need only specify a base class type.

An ADI for a class would not include a description of the encoding for primitive types or the stream tags and control information used by the `IStreamIO` class. It would also not include the description for any embedded classes; these would be obtained directly from the description of the embedded class.

Applications which need to interoperate with other applications that are not built with VisualAge may want to support a data format that does not use Open Class streaming. For example, a word processor would probably provide a way to save or export a plain-text file for exchange with other applications.

### RELATED CONCEPTS

“Chapter 1. IBM Open Class Streaming Classes” on page 1

### RELATED TASKS

“Stream Data” on page 5

“Add Streaming Support to Structs and Simple Classes”

“Create a Streamable Class” on page 8

## Add Streaming Support to Structs and Simple Classes

Structs or simple classes may be given limited streaming capabilities even though they don't inherit from `IMStreamable`. To add simple streaming support to a class:

1. If the class will be subclassed, add the functions `readFromStream()` and `writeToStream()`. These functions need not be virtual, thus the “class” need not have a vtable. The functions are similar to the corresponding functions for `IMStreamable` classes, but they may not declare a stream frame.
1. Add the streaming operators for the class.

```
YourClass::operator>>=(IDataStream& toWhere) const;  
YourClass::operator<<=(IDataStream& fromWhere);
```

In the implementation of these operators, simply invoke the corresponding `readFromStream` or `writeToStream` function for the class. You can also put the streaming functionality directly into the streaming operators, but using `readFromStream` and `writeToStream` is more consistent with objects provided by Open Class.

Instances of such classes are streamed using the streaming operators only. They may not be streamed using `writeObject` and `readObject`, they may not have stream in/out frames for RRDC, and they may not appear in contexts for detection of pointer aliases.

Alternatively, put the streaming functionality directly into global streaming operators, and omit `readFromStream()` and `writeToStream()` altogether. This approach is appropriate for classes for which source changes are not possible. It

has the disadvantages that the streaming functionality is not bundled with the class being streamed and that the structure of the code is less like that of an `IMStreamable` class. The alternative global streaming operators have this form:

```
operator>>=(const YourClass& theObject, IDataStream& toWhere);  
operator<<=(YourClass& theObject, IDataStream& fromWhere);
```

#### RELATED CONCEPTS

“Chapter 1. IBM Open Class Streaming Classes” on page 1

#### RELATED TASKS

“Create a Streamable Class”

“Create a Streamable Template Class” on page 10

“Instantiate a Stream Module” on page 4

“Instantiate a Data Stream” on page 3

“Stream Data” on page 5

“Stream Base Classes” on page 6

## Create a Streamable Class

`IMStreamable` is a virtual base class that must be mixed into all classes that support polymorphic streaming. It introduces virtual read-from and write-to stream functions that all subclasses must override. To create a streamable class, you must derive from `IMStreamable` and implement the streaming functions.

`IStreamModule` provides a way to group classes together or to qualify a class name. It provides a context for objects on a stream to distinguish a class in one DLL from a class with the same name in another DLL. Typically you will define one stream module for each DLL or application. This will eliminate name conflicts and avoid cross-DLL dependencies.

Every `IMStreamable` class, as part of its definition, must specify the stream module to which it belongs.

`StreamableDeclarationsMacro` must appear as the first item inside the declaration of all concrete classes that derive from `IMStreamable`. Abstract classes deriving from `IMStreamable` must not include this macro.

`StreamableDefinitionsMacro` must appear in the implementation `.cpp` file for all concrete classes that derive from `IMStreamable`. It identifies the `IStreamModule` to which the class belongs. Abstract classes deriving from `IMStreamable` must not include this macro.

To declare and define a streamable class, including its streaming functions, follow these steps:

1. Derive from `IMStreamable` using the macro `IMSTREAMABLE`.
  2. Override the streaming functions `writeToStream` and `readFromStream`.
  3. Associate the class with the stream module to which the class belongs.
  4. Implement the stream-out function for your class.
  5. Implement the stream-in function for your class.
1. Derive from `IMStreamable` using the macro `IMSTREAMABLE`. This macro will expand to the form of `IMStreamable` appropriate to the compiler platform.

In your class header file, include the declaration of `IMSTREAMABLE`:

```
#include <imstrmb1.hpp>
```

Declare your class as a subclass of IMSTREAMABLE:

```
class MyClass : public IMyBase, public IMSTREAMABLE {
    StreamableDeclarationsMacro(IMyClass);
private:
    // This is class instance data that must be
    // streamed by the streaming operators.
    int          fData;
    ISomeType*   fObjectPointer;
    IAnotherType* fAnotherObjectPointer;
    IBaseType*   fAbstractBasePointer;
    ISomeType    fEmbeddedObject;
    short        fShortArray[123];
    ISomeType    fSecondVersionObject;
public:
    MyClass(); // Normal user class ...
    virtual MyClass();
```

2. Override the streaming functions `writeToStream` and `readFromStream` with protected functions. The functions are invoked from the outside via `IMStreamable::operator >>=()` and `<<=()`:

```
protected:
    virtual void writeToStream(IDataStream& toWhere) const;
    virtual void readFromStream(IDataStream& fromWhere);
};
```

3. Associate the class with the stream module to which the class belongs. You must define the module itself as a global variable in one of the implementation files for your program. This macro will reference the module as an extern.

Add the following line to your `.cpp` file:

```
StreamableDefinitionsMacro(IMyClass, gMyStreamModule);
```

4. Implement the stream-out function for your class:

```
void
IMyClass::writeToStream(IDataStream &toWhere) const
{
    // Stream Frame local variable. Scope must enclose the writing of all
    // data by this stream out operator.
    IStreamOutFrame myFrame(toWhere);
    // Stream out the base class.
    // Base class streaming is the one case where
    // writeToStream is used directly, instead of operator >>=
    IMyBase::writeToStream(toWhere);
    // Stream out the data fields.
    fData >>= toWhere; // A primitive type (an int).
    *fObjectPointer >>= toWhere; // fObjectPointer must refer to
                                // a valid object. This streams the
                                // object monomorphically.
    // Stream this object monomorphically. The type parameter is
    // the actual type of the object being streamed.
    // Handle nil pointers correctly.
    ::writeObject(fAnotherObjectPointer, toWhere);
    // Always stream this object polymorphically.
    ::writeObject(fAbstractBasePointer, toWhere);
    // Embedded objects always use the streaming operators.
    // They don't need, and can't use, the nil pointer test
    // and heap allocation performed by read or write object.
    fEmbeddedObject >>= toWhere;
    // Write the array of shorts.
    toWhere.writeShorts(fShortArray, 123);
    // This object was added to the stream format after the initial
    // release.
    fSecondVersionObject >>= toWhere;
};
```

5. Implement the stream-in function for your class:

```

void
IMyClass::readFromStream(IDataStream &fromWhere)
{
    // Create a local instance of IStreamInFrame.
    IStreamInFrame myStreamFrame(fromWhere);
    // Stream in the base class.
    IMyBase::readFromStream(fromWhere);
    // Stream in the data fields.
    fData <<= fromWhere; // A primitive type (an int).
    *fObjectPointer <<= fromWhere; // fObjectPointer must refer
                                   // to a valid object.
    // Monomorphically stream in an object.
    // Handle nil pointers correctly.
    delete fAnotherObjectPointer;
    ::readObject(fAnotherObjectPointer, fromWhere,
                 IAnotherType::staticTypeRepresentation);
    // Polymorphic readObject. Stream always specifies
    // the actual type of object to read.
    delete fAbstractBasePointer;
    ::readObject(fAbstractBasePtr, fromWhere);
    fEmbeddedObject <<= fromWhere;
    // Read the array of shorts. Must know the size and have
    // storage available before calling any array read function.
    fromWhere.readShorts(fShortArray, 123);
    // fSecondVersionObject may or may not be on the stream
    // depending on which version of the class wrote to the
    // stream.
    if (!myStreamFrame.atEnd()) {
        fSecondVersionObject <<= fromWhere;
    }
};

```

#### RELATED CONCEPTS

“Chapter 1. IBM Open Class Streaming Classes” on page 1

#### RELATED TASKS

“Create a Streamable Template Class”

“Instantiate a Stream Module” on page 4

## Create a Streamable Template Class

Streamable template classes differ from ordinary non-template classes in their use of the streamable definitions and declarations macros. The macros used for ordinary classes will not expand correctly within template classes. They are replaced by a combination of inline code and an alternate declarations macro, as shown in the following example:

```

template <class AType>
class IMyClass : public IMSTREAMABLE {
    static IStreamableHelperFor< IMyClass<AType> > fgStreamableHelper;
    StreamableTemplateDeclarationsMacro;
private:
    // ...
};

```

Each instantiation of a template class is really a new class type. When you instantiate a streamable template class, you must define and initialize its `IStreamableHelperFor` variable:

The following shows an instantiation of `IMyClass` over `int` in the program’s implementation file:

```

#include <myclass.hpp>
extern IStreamModule gMyStreamModule;
IMyClass<int> foo;
IStreamableHelperFor< IMyClass<int> > IMyClass<int>::fgStreamableHelper(
    gMyStreamModule, 'IMyClass<int>');

```

The following shows another instantiation of the IMyClass, this time over IString:

```

#include <myclass.hpp>
extern IStreamModule gMyStreamModule;
IMyClass<IString> bar;
IStreamableHelperFor<IMyClass<IString> > IMyClass<int>::fgStreamableHelper(
    gMyStreamModule, 'IMyClass<IString>');

```

#### RELATED CONCEPTS

“Chapter 1. IBM Open Class Streaming Classes” on page 1  
 “Data Streams” on page 3

#### RELATED TASKS

“Create a Streamable Class” on page 8  
 “Instantiate a Stream Module” on page 4

---

## Release-to-Release Data Compatibility (RRDC)

Class version mismatches can occur when two machines running different software versions exchange data or, on a single machine, when upgraded software accesses older data. With VisualAge C++ streaming classes, however, objects that are created and written to a stream by one version of a class implementation are readable on either a newer or older version.

Release-to-release data compatibility does not make use of version numbers. Class data on a stream are bounded by a frame that allows stream in functions for a class to know when they have reached the end of the data that was written or to skip over any extra data. Thus, new implementations are able to sense that they are reading an older stream data, and do not try to read data that does not exist on the stream. Old class implementations are able to sense and skip over extra data that was written by newer versions.

RRDC can support any number of versions of a class. The main requirements are as follows:

- Each newer version must make its additions to the stream data format at the end of the data written by the previous version.
- Data expected by older versions must continue to be written by newer versions. If the new version doesn't use the old data, then at stream-out time the new version must synthesize the old data or at least a suitable approximation of it to enable old versions to work when they stream in the data.

RRDC is only available when using one of the tagged stream formats (see “Instantiate a Data Stream” on page 3 for available stream formats). Raw binary stream formats omit the tags that delimit object data on a stream. They are suitable for caching data, streaming between two threads in the same process, and for streaming between two processes on the same machine when both processes are using the same version of a class. Raw binary stream formats are not suitable for persistent data or for streaming between systems that may have different versions of the same DLLs installed. The class wishing to provide RRDC can not enforce the use of a tagged stream format; it's up to whoever specifies the stream to choose an appropriate encoding. RRDC enabled classes (those containing

IStreamIn/OutFrames) will stream their data correctly on non-tagged stream; it's only the ability to detect the end of a frame on reading that is lost.

Additional stream frames can be used within a class to delimit variable length data.

#### RELATED CONCEPTS

“Chapter 1. IBM Open Class Streaming Classes” on page 1

#### RELATED TASKS

“Stream Data” on page 5

“Instantiate a Data Stream” on page 3

“Enable Release-to-Release Data Compatibility”

“Migrate from Older Streaming Constructs” on page 14

## Enable Release-to-Release Data Compatibility

To enable RRDC for a class:

1. The streaming functions must declare a local (stack) instance of a stream frame object, the scope of which surrounds the actual streaming of the data. This is done with IStreamInFrame and IStreamOutFrame. For a writeToStream function, the frame will cause the extent of the object's data on the stream to be tagged. For a readFromStream function, the frame will facilitate testing for the end of the object's data and skipping over data to reach the end of the object.
2. On stream out, any extension data (additional data added to the stream format by a new release of the class) must be streamed after the original data.
3. On stream in, a test to verify that the end of the stream data for that object has not been reached must precede the reading of any extension data. This will ensure that the stream was not written by an old class implementation that did not provide extension data. On stream in, if the object's data on the stream has not been exhausted at the end of the readFromStream function, as signaled by the stream frame object going out of scope, the remaining unread data will be skipped.

RRDC is only available when one of the tagged stream formats is used. Raw binary stream formats, which omit the tags that delimit object data on a stream, are not suitable for persistent data or for streaming between systems that may have different libraries installed. It is up to the client that specifies the stream to choose an appropriate encoding. RRDC enabled classes will stream out their data correctly on nontagged streams, but they will be unable to detect the end of a frame on streaming in.

In the following example, the fData variable is replaced, in a later version of the class, by fOldData and fNewData.

Here is the original version of DerivedClass::writeToStream:

```
void
DerivedClass::writeToStream(IDataStream& toStream) const
{
    IStreamOutFrame streamContext(toStream); // Must be declared on stack.
    BaseClass::writeToStream(toStream);     // Write base class data first.
    fData >>= toStream;                     // Write instance data to stream.
}
```

This is the original version of DerivedClass::readFromStream:

```

void
DerivedClass::readFromStream(IDataStream& fromStream)
{
    IStreamInFrame streamContext(fromStream); // Must be declared on stack
    BaseClass::readFromStream(fromStream); // Read base class data first.
    fData <<= fromStream; // Read instance data from stream.
}

```

This is the new version of DerivedClass::writeToStream:

```

void
DerivedClass::writeToStream(IDataStream& toStream) const
{
    IStreamOutFrame streamContext(toStream);
    BaseClass::writeToStream(toStream);
    fOldData >>= toStream; // Used to be fData
    fNewData >>= toStream;
}

```

This is the new version of DerivedClass::readFromStream:

```

void
DerivedClass::readFromStream(IDataStream& fromStream) const
{
    IStreamInFrame streamContext(fromStream);
    BaseClass::readFromStream(fromStream);
    fOldData <<= fromStream;
    // fNewData may or may not be on the stream depending on
    // which version of the class wrote to the stream.
    if (!streamContext.atEnd()) {
        fNewData <<= fromWhere;
    }
    else {
        fNewData = fOldData; // The new data wasn't there so
        // use the old data.
    }
}

```

#### RELATED CONCEPTS

“Chapter 1. IBM Open Class Streaming Classes” on page 1  
“Release-to-Release Data Compatibility (RRDC)” on page 11

#### RELATED TASKS

“Migrate from Older Streaming Constructs” on page 14

---

## Exceptions Defined by the Streaming Classes

Streaming defines the following exception classes:

- IInvalidDataOnStream
- IUnknownTypeOnStream
- IEndOfStream
- IAddressAlreadyInContext
- IInvalidContextNumber

These exceptions can be raised while reading from a stream. They generally indicate that the stream data is corrupt or that the type of data written does not correspond to the type of data that the program is attempting to read from the stream.

#### RELATED CONCEPTS

“Chapter 1. IBM Open Class Streaming Classes” on page 1  
 Exceptions in the IBM Open Class

**RELATED TASKS**

“Instantiate a Data Stream” on page 3

## Migrate from Older Streaming Constructs

This table compares the old streaming constructs with the new constructs:

Old streaming construct	New streaming construct
<pre>obj_ptr2 = ::copyPointer(obj_ptr1) obj_ptr2 = ::copy(src)</pre>	<pre>obj_ptr2 = ::copyPointer(obj_ptr1); obj_ptr2 = ::copy(src)</pre> <p>No Change.</p>
<pre>primitiveType &lt;&lt;= aStream primitiveType &gt;&gt;= aStream</pre>	<pre>primitiveType &lt;&lt;= aStream primitiveType &gt;&gt;= aStream</pre> <p>Streaming of primitive types is unchanged.</p>
<pre>object_reference &gt;&gt;= aStream object_reference &lt;&lt;= aStream</pre>	<pre>object_reference &gt;&gt;= aStream object_reference &lt;&lt;= aStream</pre> <p>Monomorphic streaming of embedded objects is unchanged.</p>
<pre>Class IFoo {     ... }</pre>	<pre>class IFoo : public virtual IStreamable {     ... }</pre> <p>Streamable classes must inherit from IStreamable.</p>
TypeExtension[*]Macro	<pre>StreamableDeclarationsMacro StreamableDefinitionsMacro</pre> <p>No macros for enabling dynamic cast or copy.</p>
<pre>yourClass::operator&lt;&lt;=(IBaseStream&amp;) yourClass::operator&gt;&gt;=(IBaseStream&amp;)</pre>	<pre>yourClass::readFromStream(IDataStream&amp;) yourClass::writeToStream(IDataStream&amp;)</pre> <p>User written streaming functions for a class.</p>
readVersion(aStream)	IStreamInFrame localFrame(aStream);
writeVersion(aStream, version_number)	<pre>IStreamOutFrame localFrame(aStream);</pre> <p>Handling of versions is substantially different.</p>



Old streaming construct	New streaming construct
<pre>BaseClass::operator&gt;&gt;=(aStream) BaseClass::operator&lt;&lt;=(aStream)</pre>	<pre>BaseClass::writeToStream(aStream); BaseClass::readFromStream(aStream);</pre> <p>Streaming base classes from within a derived class streaming operator.</p>
<pre>:flatten(objectPtr, aStream)</pre>	<pre>aStream.writeObject(objectPtr);</pre>
<pre>::resurrect(objectPtr, aStream)</pre>	<pre>::readObject(objectPtr, aStream);</pre>
<pre>dynamicCastTo(targ_ptr, src_ptr)</pre>	<pre>targ_ptr = dynamic_cast&lt;ITargetType*&gt;(src_ptr)</pre> <p>This is standard C++.</p>
<pre>createNewObject</pre>	<p>Not available.</p>
<pre>class IBaseStream</pre>	<pre>class IDataStream</pre> <p>Subclassing interface is different. Random access is omitted. Construction is different.</p>
<pre>IBaseStream::SetContext()</pre>	<pre>IStreamContextFrame(IDataStream&amp;);</pre> <p>Local object to establish a context for top-level users of a stream.</p>

#### RELATED CONCEPTS

“Chapter 1. IBM Open Class Streaming Classes” on page 1  
“Release-to-Release Data Compatibility (RRDC)” on page 11

#### RELATED TASKS

“Add Streaming Support to Structs and Simple Classes” on page 7  
“Create a Streamable Class” on page 8  
“Instantiate a Data Stream” on page 3  
“Enable Release-to-Release Data Compatibility” on page 12



---

## Chapter 2. I/O Stream Classes

This section refers to the non-ANSI I/O Stream Library.

VisualAge C++ comes with ANSI-compliant stream classes. We recommend that you use these stream classes to develop thread-safe applications. The non-ANSI I/O Stream Library is not thread-safe. The ANSI-compliant stream classes are part of the Standard C++ Library. For more information about the standard C++ I/O stream library see Chapter 21 entitled “Streams” of Bjarne Stroustrup’s *The C++ Programming Language*. This reference is included with the VisualAge C++ PDF files.

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

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

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

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

### Input and Output for User-Defined Classes

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

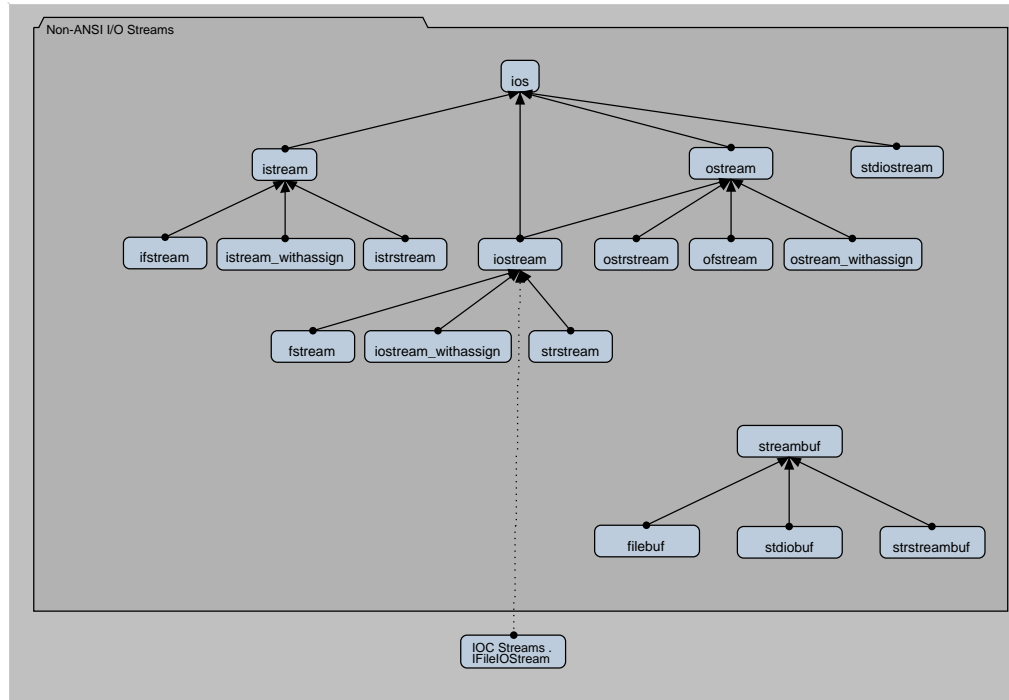
#### RELATED CONCEPTS

The I/O Stream Class Hierarchy  
I/O Stream Header Files  
Stream Buffers  
Format State Flags  
Manipulators  
The I/O Stream Classes and `stdio.h`

#### RELATED REFERENCES

## The I/O Stream Class Hierarchy

The I/O Stream Classes have two base classes, `streambuf` and `ios`:



The `streambuf` class implements *stream buffers*. `streambuf` is the base class for the following classes:

- `strstreambuf`
- `stdiobuf`
- `filebuf`

The `ios` class maintains formatting and error state information for streams. Streams are implemented as objects of the following classes that are derived from `ios`:

- `stdiostream`
- `istream`
- `ostream`

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

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

- `istrstream`
- `ifstream`
- `istream_withassign`
- `iostream`

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

- `ostrstream`

- ofstream
- ostream\_withassign
- iostream

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

- stringstream
- ostream\_withassign
- fstream

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

#### RELATED CONCEPTS

I/O Stream Classes

---

## I/O Stream Header Files

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



The header file iostream.h contains declarations for the basic classes:

- stringstreambuf
- 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 stdiofilebuf and stdiostream, the classes that specialize stringstreambuf 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.

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

  The 8.3 file naming convention compliant name of this file is stringstr.h.

The first “str” in each of these names stands for “string”:

- istrstream
- ostrstream

- `stringstream`
- `stringstreambuf`

The header file `iosmanip.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 I/O Stream Classes. It includes `iostream.h`, `fstream.h`, `fstream.h`, and `iosmanip.h`, along with some definitions needed for compatibility with the AT&T C++ Language System Release 1.2. Only use this header file with existing code; do not use it with new C++ code.

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

#### RELATED CONCEPTS

I/O Stream Classes

---

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

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

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

#### RELATED CONCEPTS

I/O Stream Classes

---

## Use Predefined Streams

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

- `cin` is the standard input stream.
  - ▶ **AIX** file descriptor = 0.
- `cout` is the standard output stream.
  - ▶ **AIX** 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 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 file descriptor = 2.

AIX

WIN OS/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.

#### RELATED CONCEPTS

I/O Stream Classes

#### RELATED TASKS

- “Combine Input and Output of Different Types” on page 46
- “Receive Input from Standard Input” on page 36
- “Display Output on Standard Output or Standard Error” on page 38
- “Flush Output Streams with `endl` and `flush`” on page 40
- “Parse Multiple Inputs” on page 42
- “Open and Read from Files” on page 43
- “Open and Write to Files” on page 46
- “Associate a File with a Standard Input or Output Stream” on page 47
- “Move through a file with `filebuf` Functions” on page 48
- “Define an Input Operator for a Class Type” on page 51
- “Define an Output Operator for a Class Type” on page 53
- “Correct Input Stream Errors” on page 55
- “Format Stream Output” on page 23
- “Define Your Own Format State Flags” on page 28
- “Manipulate Strings with the `stringstream` Classes” on page 58
- “Create Manipulators” on page 32

---

## Stream Buffers

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

The AT&T and UNIX<sup>®</sup> System Laboratories C++ Language System documentation use the terms *reserve area* and *buffer* instead of *stream buffer*.

### What Does a Stream Buffer Do?

A stream buffer acts as a buffer between the *ultimate producer* (the source of data) or *ultimate consumer* (the target of data) and the member functions of the classes

derived from ios that format this raw data. The ultimate producer can be a file, a device, or an array of bytes in memory. The ultimate consumer can also be a file, a device, or an array of bytes in memory.

### Why Use a Stream Buffer?

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

Consider, for example, an application that is reading data from the ultimate producer. If there is no buffer, the application has to make a system call for each character that is read. However, if the application uses a stream buffer, system calls will only be made when the buffer is empty. Each system call will read enough characters from the ultimate producer (if they are available) to fill the buffer again.

### How is a stream buffer implemented?

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

A stream buffer does not necessarily have separate get and put areas:

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

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

Member function	Description
base	Returns a pointer to the beginning of the stream buffer.
eback	Returns a pointer to the beginning of the space available for <i>putback</i> . Characters that are putback are returned to the get area of the stream buffer.
gptr	Returns the <i>get pointer</i> (a pointer to the beginning of the get area). The space between gptr and egptr has been filled by the ultimate processor.
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.



Member function	Description
pptr	Returns the <i>put pointer</i> (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.

#### RELATED CONCEPTS

I/O Stream Classes

---

## Format State Flags

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

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

#### RELATED CONCEPTS

I/O Stream Classes

## Format Stream Output

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

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

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

### 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;
}
void main () {
// get and display current flag settings using flags()
cout << "flags(): ";
long originalFlags = cout.flags();
showFlags();
// change format state using flags(long)
cout << "flags(ios::fixed): ";
cout.flags(ios::fixed);
showFlags();
// change adjust field using setf(long)
cout << "setf(ios::right): ";
cout.setf(ios::right);
showFlags();
// change floatfield using setf(long, long)
cout << "setf(ios::scientific | ios::left,\n"
<< "ios::floatfield | ios::adjustfield): ";
cout.setf(ios::scientific | ios::left,ios::floatfield | ios::adjustfield);
showFlags();
// reset to original setting
cout << "flags(originalFlags): ";
cout.flags(originalFlags);
showFlags();
}
```

This example produces the following output:

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

Note 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>
void main() {
    double a=3.14159265358979323846;
    double b;
    long originalFlags=cout.flags();
    int originalPrecision=cout.precision();
    for (double exp=1.;exp<1.0E+25;exp*=100000000.) {
        cout << "Printing new value for b:\n";
        b=a*exp; // Initialize b to a larger magnitude of a
    }
    // Now print b in a number of ways:
    // In fixed decimal notation
    cout.setf(ios::fixed,ios::floatfield);
    cout << " " << b << '\n';
    // In scientific notation
    cout.setf(ios::scientific,ios::floatfield);
    cout << " " << b << '\n';
    // Change the exponent from lower to uppercase
    cout.setf(ios::uppercase);
    cout << " " << b << '\n';
    // With 12 digits of precision, scientific notation
    cout.precision(12);
    cout << " " << b << '\n';
    // Same precision, fixed notation
    cout.setf(ios::fixed,ios::floatfield);
    // Now set everything back to defaults
    cout.flags(originalFlags);
    cout.precision(originalPrecision);
}
```

The output from this program is:

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

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

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

### 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()
void main() {
    int i=-5,j=7,k=-9;
    cout << setw(5) << i << setw(5) << j << setw(5) << k << endl;
}
```

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

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

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

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

#### RELATED CONCEPTS

I/O Stream Classes

#### RELATED TASKS

- “Combine Input and Output of Different Types” on page 46
- “Use Predefined Streams” on page 20
- “Receive Input from Standard Input” on page 36
- “Display Output on Standard Output or Standard Error” on page 38
- “Flush Output Streams with `endl` and `flush`” on page 40
- “Parse Multiple Inputs” on page 42
- “Open and Read from Files” on page 43
- “Open and Write to Files” on page 46
- “Associate a File with a Standard Input or Output Stream” on page 47
- “Move through a file with `filebuf` Functions” on page 48
- “Define an Input Operator for a Class Type” on page 51
- “Define an Output Operator for a Class Type” on page 53
- “Correct Input Stream Errors” on page 55
- “Define Your Own Format State Flags”
- “Manipulate Strings with the `stringstream` Classes” on page 58
- “Create Manipulators” on page 32

## 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 I/O Stream Classes let you define stream-specific flags that you can

then use with the format state member functions such as `setf()` and `unsetf()`. You can then code checks for these flags in the input and output operators you write for your class types, and determine how to handle input and output according to the settings of those flags.

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

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

```
// Defining your own format flags
#include <fstream.h>
#include <stdlib.h>
long InFileFormat=0;
long OutFileFormat=0;
class CustRecord {
public:
    int Number;
    char Name[48];
    char Phone[16];
    char Street[128];
    char City[64];
    char Country[64];
    char PostCode[10];
};
ostream& operator<<(ostream &os, CustRecord &cust) {
    os << cust.Number << '\n'
    << cust.Name << '\n'
    << cust.Phone << '\n'
    << cust.Street << '\n'
    << cust.City << '\n';
    if (os.flags() & OutFileFormat) // New file format
        os << cust.Country << '\n'
        << cust.PostCode << endl;
    else // Old file format
        os << cust.PostCode << '\n'
        << cust.Country << endl;
    return os;
}
istream& operator>>(istream &is, CustRecord &cust) {
    is >> cust.Number;
    is.ignore(1000, '\n'); // Ignore anything up to and including new line
    is.getline(cust.Name,48);
    is.getline(cust.Phone,16);
    is.getline(cust.Street,128);
    is.getline(cust.City,64);
    if (is.flags() & InFileFormat) { // New file format!
        is.getline(cust.Country,64);
        is.getline(cust.PostCode,10);
    }
    else {
        is.getline(cust.PostCode,10);
    }
}
```

```

        is.getline(cust.Country,64);
    }
    return is;
}
void main(int argc, char* argv[]) {
    if (argc!=3) { // Requires two parameters
        cerr << "Specify an input file and an output file\n";
        exit(1);
    }
    ifstream InFile(argv[1]);
    ofstream OutFile(argv[2],ios::out);
    InFileFormat = InFile.bitalloc(); // Allocate flags for
    OutFileFormat = OutFile.bitalloc(); // each fstream
    if (InFileFormat==0 || // Exit if no flag could
        OutFileFormat==0) { // be allocated
        cerr << "Could not allocate a user-defined format flag.\n";
        exit(2);
    }
    if (InFile.peek()=='!') { // '!' means new format
        InFile.setf(InFileFormat); // Input file is in new format
        OutFile.unsetf(OutFileFormat); // Output file is in old format
        InFile.get(); // Clear that first byte
    }
    else { // Otherwise, write '!' to
        OutFile << '!'; // the output file, set the
        OutFile.setf(OutFileFormat); // output stream's flag, and
        InFile.unsetf(InFileFormat); // clear the input stream's
    } // flag

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

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

```

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

Input File	Output File
3848 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.

#### RELATED CONCEPTS

I/O Stream Classes

#### RELATED TASKS

“Combine Input and Output of Different Types” on page 46  
“Use Predefined Streams” on page 20  
“Receive Input from Standard Input” on page 36  
“Display Output on Standard Output or Standard Error” on page 38  
“Flush Output Streams with endl and flush” on page 40  
“Parse Multiple Inputs” on page 42  
“Open and Read from Files” on page 43  
“Open and Write to Files” on page 46  
“Associate a File with a Standard Input or Output Stream” on page 47  
“Move through a file with filebuf Functions” on page 48  
“Define an Input Operator for a Class Type” on page 51  
“Define an Output Operator for a Class Type” on page 53  
“Correct Input Stream Errors” on page 55  
“Format Stream Output” on page 23  
“Manipulate Strings with the stringstream Classes” on page 58  
“Create Manipulators” on page 32

---

## 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 ostream library operations.

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

### Simple Manipulators and Parameterized Manipulators

There are two kinds of manipulators: *simple* 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 `iosmanip.h` header file) is an example of a parameterized manipulator. You can create your own parameterized manipulators and your own simple manipulators.

### ios Methods and Manipulators

For some of the format flags defined for the `ios` class, you can set or clear them using an `ios` function and a flag name, or by using a manipulator. With manipulators you can place the change to a stream's state within a list of outputs for that stream.

#### RELATED CONCEPTS

I/O Stream Classes

## Create Manipulators

### Create Simple Manipulators for Your Own Types

The I/O Stream Class 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
<code>istream</code>	<code>istream &amp;fi(istream&amp;){ /*...*/ }</code>
<code>ostream</code>	<code>ostream &amp;fo(ostream&amp;){ /*...*/ }</code>
<code>ios</code>	<code>ios &amp;fios(ios&amp;){ /*...*/ }</code>

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 I/O Stream Class 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
<code>istream</code>	<code>IMANIP(tp)</code> or <code>IAPP(tp)</code>
<code>ostream</code>	<code>OMANIP(tp)</code> or <code>OAPP(tp)</code>
<code>iostream</code>	<code>IOMANIP(tp)</code> or <code>IOAPP(tp)</code>
The <code>ios</code> part of <code>istream</code> objects or <code>ostream</code> objects	<code>SMANIP(tp)</code> or <code>SAPP(tp)</code>

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

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

4. Define the manipulator.

Parameterized manipulators defined with `IOMANIP` or `IOAPP` are not associative. This means that you cannot use such manipulators more than once in a single output statement.

#### RELATED CONCEPTS

I/O Stream Classes

#### RELATED TASKS

- “Define an APP Parameterized Manipulator” on page 34
- “Define a MANIP Parameterized Manipulator” on page 34
- “Define Nonassociative Parameterized Manipulators” on page 35

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

This program produces the following output:

——Hello

### RELATED CONCEPTS

I/O Stream Classes

### RELATED TASKS

“Create Manipulators” on page 32

“Define a MANIP Parameterized Manipulator”

“Define Nonassociative Parameterized Manipulators” on page 35

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

```

This example produces the following output:

——Hello

#### RELATED CONCEPTS

I/O Stream Classes

#### RELATED TASKS

“Create Manipulators” on page 32

“Define an APP Parameterized Manipulator” on page 34

“Define Nonassociative Parameterized Manipulators”

## 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>
ostream& f(ostream & io, int i) {
    io.width(i);
    return io;
}
IOMANIP (int) mysetw(int i) {
    return IOMANIP(int) (f,i);
}
ostream_withassign ioswa;
void main() {
    ioswa = cout;
    int i1 = 8, i2 = 14;
    //
    // The following statement does not cause a compile-time
    // error.
    //
    ioswa << mysetw(3) << i1 << endl;
    //
    // The following statement causes a compile-time error
    // because the manipulator mysetw is applied twice.
    //
    ioswa << mysetw(3) << i1 << mysetw(5) << i2 << endl;
}

```

```

//
// 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;
}

```

#### RELATED CONCEPTS

I/O Stream Classes

#### RELATED TASKS

“Create Manipulators” on page 32

“Define an APP Parameterized Manipulator” on page 34

“Define a MANIP Parameterized Manipulator” on page 34

## Basic 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.

An input operator must exist for the type being read in. The I/O Stream Class 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.

#### 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:

#### String Input

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

```

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

```

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

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

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

### White Space in String Input

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

```

// String input using operator << and getline()
#include <iostream.h>
void main() {
    char text1[100], text2[100];
    // prompt and get input for text arrays
    cout << "Enter two words:\n";
    cin >> text1 >> text2;

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

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

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

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

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

```

The first argument of `getline()` is a pointer to the character array in which to store the input. The second argument specifies the maximum number of bytes of input to read 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.

#### **RELATED CONCEPTS**

I/O Stream Classes

#### **RELATED TASKS**

- “Combine Input and Output of Different Types” on page 46
- “Use Predefined Streams” on page 20
- “Display Output on Standard Output or Standard Error”
- “Flush Output Streams with `endl` and `flush`” on page 40
- “Parse Multiple Inputs” on page 42
- “Open and Read from Files” on page 43
- “Open and Write to Files” on page 46
- “Associate a File with a Standard Input or Output Stream” on page 47
- “Move through a file with `filebuf` Functions” on page 48
- “Define an Input Operator for a Class Type” on page 51
- “Define an Output Operator for a Class Type” on page 53
- “Correct Input Stream Errors” on page 55
- “Format Stream Output” on page 23
- “Define Your Own Format State Flags” on page 28
- “Manipulate Strings with the `stringstream` Classes” on page 58
- “Create Manipulators” on page 32

## **Display Output on Standard Output or Standard Error**

The I/O Stream Class 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>
void main(int argc, char* argv[]) {
    if (argc==1) cout << "Good day!" << endl;
    else cerr << "I don't know what to do with "
        << argv[1] << endl;
}
```

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

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

An output operator must exist for any type being output. The I/O Stream Class 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;
```

#### RELATED CONCEPTS

I/O Stream Classes

#### RELATED TASKS

- "Combine Input and Output of Different Types" on page 46
- "Use Predefined Streams" on page 20
- "Receive Input from Standard Input" on page 36
- "Flush Output Streams with endl and flush"
- "Parse Multiple Inputs" on page 42
- "Open and Read from Files" on page 43
- "Open and Write to Files" on page 46
- "Associate a File with a Standard Input or Output Stream" on page 47
- "Move through a file with filebuf Functions" on page 48
- "Define an Input Operator for a Class Type" on page 51
- "Define an Output Operator for a Class Type" on page 53
- "Correct Input Stream Errors" on page 55
- "Format Stream Output" on page 23
- "Define Your Own Format State Flags" on page 28
- "Manipulate Strings with the stringstream Classes" on page 58
- "Create Manipulators" on page 32

## 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.

## RELATED CONCEPTS

I/O Stream Classes

## RELATED TASKS

- “Combine Input and Output of Different Types” on page 46
- “Use Predefined Streams” on page 20
- “Receive Input from Standard Input” on page 36
- “Display Output on Standard Output or Standard Error” on page 38
- “Flush Output Streams with endl and flush” on page 40
- “Parse Multiple Inputs”
- “Open and Read from Files” on page 43
- “Open and Write to Files” on page 46
- “Associate a File with a Standard Input or Output Stream” on page 47
- “Move through a file with filebuf Functions” on page 48
- “Define an Input Operator for a Class Type” on page 51
- “Define an Output Operator for a Class Type” on page 53
- “Correct Input Stream Errors” on page 55
- “Format Stream Output” on page 23
- “Define Your Own Format State Flags” on page 28
- “Manipulate Strings with the stringstream Classes” on page 58
- “Create Manipulators” on page 32

## Parse Multiple Inputs

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

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

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

Input	Output	Remarks
123<\n>		No output. a has been assigned the value 123, but the program is still waiting on input for b and c.
1<\n> ,br 2<\n> 3<\n>	a: <1> b: <2> c: <3>	White space (in this case, new-line characters) is used to delimit different input variables.
1 2 3<\n>	a: <1> b: <2> c: <3>	White space (in this case, spaces) is used to delimit different input variables. There can be any amount of white space between inputs.

Input	Output	Remarks
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.

#### RELATED CONCEPTS

I/O Stream Classes

#### RELATED TASKS

- “Combine Input and Output of Different Types” on page 46
- “Use Predefined Streams” on page 20
- “Receive Input from Standard Input” on page 36
- “Display Output on Standard Output or Standard Error” on page 38
- “Flush Output Streams with endl and flush” on page 40
- “Open and Read from Files”
- “Open and Write to Files” on page 46
- “Associate a File with a Standard Input or Output Stream” on page 47
- “Move through a file with filebuf Functions” on page 48
- “Define an Input Operator for a Class Type” on page 51
- “Define an Output Operator for a Class Type” on page 53
- “Correct Input Stream Errors” on page 55
- “Format Stream Output” on page 23
- “Define Your Own Format State Flags” on page 28
- “Manipulate Strings with the stringstream Classes” on page 58
- “Create Manipulators” on page 32

## Open and Read from Files

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.
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>
void main() {
    fstream infile1;
    ifstream infile2;
```

```

        infile1.open("myfile.dat",ios::in);
        infile2.open("myfile.dat");
        // ...
    }

```

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

```

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

```

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

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

```

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

```

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

```
barbers often shave
```

the file contains the following after the program is run:

```
barbers often shave
barbers
```

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

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

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

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

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

All techniques for reading input from the standard input stream can also be used to read input from a file, providing your code is changed so that the cin object is replaced with the name of the fstream object associated with the input file.

#### RELATED CONCEPTS

I/O Stream Classes

#### RELATED TASKS

- “Combine Input and Output of Different Types” on page 46
- “Use Predefined Streams” on page 20
- “Receive Input from Standard Input” on page 36
- “Display Output on Standard Output or Standard Error” on page 38
- “Flush Output Streams with endl and flush” on page 40
- “Parse Multiple Inputs” on page 42
- “Open and Write to Files” on page 46
- “Associate a File with a Standard Input or Output Stream” on page 47
- “Move through a file with filebuf Functions” on page 48
- “Define an Input Operator for a Class Type” on page 51
- “Define an Output Operator for a Class Type” on page 53
- “Correct Input Stream Errors” on page 55
- “Format Stream Output” on page 23
- “Define Your Own Format State Flags” on page 28
- “Manipulate Strings with the strstream Classes” on page 58
- “Create Manipulators” on page 32

## Open and Write to Files

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

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

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

You must specify one or more open modes when you open the file, unless you declare the object as an `ofstream` object. The advantage of accessing an output file as an `ofstream` object rather than as an `fstream` object is that the compiler can flag input operations to that object as errors.

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

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

### RELATED CONCEPTS

I/O Stream Classes

### RELATED TASKS

“Combine Input and Output of Different Types”

“Use Predefined Streams” on page 20

“Receive Input from Standard Input” on page 36

“Display Output on Standard Output or Standard Error” on page 38

“Flush Output Streams with `endl` and `flush`” on page 40

“Parse Multiple Inputs” on page 42

“Open and Read from Files” on page 43

“Associate a File with a Standard Input or Output Stream” on page 47

“Move through a file with `filebuf` Functions” on page 48

“Define an Input Operator for a Class Type” on page 51

“Define an Output Operator for a Class Type” on page 53

“Correct Input Stream Errors” on page 55

“Format Stream Output” on page 23

“Define Your Own Format State Flags” on page 28

“Manipulate Strings with the `stringstream` Classes” on page 58

“Create Manipulators” on page 32

## Combine Input and Output of Different Types

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

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

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



## RELATED CONCEPTS

I/O Stream Classes

## RELATED TASKS

“Use Predefined Streams” on page 20  
“Receive Input from Standard Input” on page 36  
“Display Output on Standard Output or Standard Error” on page 38  
“Flush Output Streams with endl and flush” on page 40  
“Parse Multiple Inputs” on page 42  
“Open and Read from Files” on page 43  
“Open and Write to Files” on page 46  
“Associate a File with a Standard Input or Output Stream”  
“Move through a file with filebuf Functions” on page 48  
“Define an Input Operator for a Class Type” on page 51  
“Define an Output Operator for a Class Type” on page 53  
“Correct Input Stream Errors” on page 55  
“Format Stream Output” on page 23  
“Define Your Own Format State Flags” on page 28  
“Manipulate Strings with the stringstream Classes” on page 58  
“Create Manipulators” on page 32

---

## Advanced 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>

void main(int argc, char* argv[]) {
    ifstream* infile;
    ofstream* outfile;
    char inputline[4096];           // used to read input lines
    int sinl=sizeof(inputline);    // used by getline() function
    if (argc>1) {                  // if at least an input file was specified
        infile = new ifstream(argv[1]); // try opening it
        if (infile->good())          // if it opens successfully
            cin = *infile;          // assign input file to cin
        if (argc>2) {              // if an output file was also specified
            outfile = new ofstream(argv[2]); // try opening it
            if (outfile->good())     // if it opens successfully
                cout = *outfile;    // assign output file to cout
        }
    }
}
```

```

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

```

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

#### RELATED CONCEPTS

I/O Stream Classes

#### RELATED TASKS

- “Combine Input and Output of Different Types” on page 46
- “Use Predefined Streams” on page 20
- “Receive Input from Standard Input” on page 36
- “Display Output on Standard Output or Standard Error” on page 38
- “Flush Output Streams with endl and flush” on page 40
- “Parse Multiple Inputs” on page 42
- “Open and Read from Files” on page 43
- “Open and Write to Files” on page 46
- “Move through a file with filebuf Functions”
- “Define an Input Operator for a Class Type” on page 51
- “Define an Output Operator for a Class Type” on page 53
- “Correct Input Stream Errors” on page 55
- “Format Stream Output” on page 23
- “Define Your Own Format State Flags” on page 28
- “Manipulate Strings with the stringstream Classes” on page 58
- “Create Manipulators” on page 32

## 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

```

```

void main() {
    // declare a streampos object to keep track of the position in filebuf
    streampos Position;
    // declare a streamoff object to set stream offsets
    // (for use by seekoff and seekpos)
    streamoff Offset=0;
    // declare an fstream object and open its file for input
    fstream InputFile("algonq.uin",ios::in);
    // check that input was successful, exit if not
    if (!InputFile) {
        cerr << "Could not open algonq.uin! Exiting...\n";
        exit(-1);
    }
    // associate the fstream object with a filebuf pointer
    filebuf *InputBuffer=InputFile.rdbuf();
    // read the first line, and display it
    char LineOfFile[128];
    InputFile.getline(LineOfFile,sizeof(LineOfFile),'\n');
    cout << LineOfFile << endl;
    // Now skip forward 100 bytes and display another line
    Offset=100;
    Position=InputBuffer->seekoff(Offset,ios::cur,ios::in);
    InputFile.getline(LineOfFile,sizeof(LineOfFile),'\n');
    cout << "At position " << Position << ":\n"
        << LineOfFile << endl;
    // Now skip back 50 bytes and display another line
    Offset=-50;
    Position=InputBuffer->seekoff(Offset,ios::cur,ios::in);
    // ios::cur refers to current position in buffer
    InputFile.getline(LineOfFile,sizeof(LineOfFile),'\n');
    cout << "At position " << Position << ":\n"
        << LineOfFile << endl;
    // Now go to position 137 and display to the end of its line
    Position=137;
    InputBuffer->seekpos(Position,ios::in);
    InputFile.getline(LineOfFile,sizeof(LineOfFile),'\n');
    cout << "At position " << Position << ":\n"
        << LineOfFile << endl;
    // Now close the file and end the program
    InputFile.close();
}

```

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

```

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

```

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

the output of the example program is:

```

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

```

### 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 OS/390<sup>®</sup> C/C++ and programs that use them may not be portable to other VisualAge C++ (formerly C Set++) compilers.

```
// Example of using encoded and relative byte offsets
// in seeking through a file
#include <iomanip.h>
#include <fstream.h>
void main() {
    fstream    fs("tseek.data", ios::out); // create tseek.data
    filebuf*   fb = fs.rdbuf();
    streamoff  off[5];
    int        pos[5] = {0, 30, 42, 197, 0};
    for (int i = 0, j = 0; i < 200; ++i) {
        if (i == pos[j])
            off[j++] = (*fb).seekoff(0L, ios::cur, ios::out);
        fs << setw(4) << i;
        if (i % 13 == 0 || i % 17 == 0) fs << endl;
    }
    fs.close();
    cout << "Open the file in text mode, reposition using encoded\n"
         << "offsets obtained from previous calls to seekoff()" << endl;
    fs.open("tseek.data", ios::in);
    fb = fs.rdbuf();
    // Exchange off[2] and off[3] so last seek will be backwards
    off[4] = off[2]; off[2] = off[3]; off[3] = off[4];
    pos[4] = pos[2]; pos[2] = pos[3]; pos[3] = pos[4];
    for (j = 0; j < 4; ++j) {
        (*fb).seekoff(off[j], ios::beg, ios::in);
        fs >> i;
        cout << "data at pos" << dec << setfill(' ') << setw(4) << pos[j]
             << " is \"" << setw(4) << i << "\" (encoded offset was 0x"
             << hex << setfill('0') << setw(8) << off[j] << ")" << endl;
        if (i != pos[j]) return 37 + 10*j;
    }
    fs.close();
    cout.fill(' ');
    cout.setf(ios::dec, ios::basefield);
    cout << "\nOpen the file in binary 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;
    }
}
```

#### RELATED CONCEPTS

I/O Stream Classes

#### RELATED TASKS

“Combine Input and Output of Different Types” on page 46

“Use Predefined Streams” on page 20

“Receive Input from Standard Input” on page 36

“Display Output on Standard Output or Standard Error” on page 38

“Flush Output Streams with endl and flush” on page 40

“Parse Multiple Inputs” on page 42  
 “Open and Read from Files” on page 43  
 “Open and Write to Files” on page 46  
 “Associate a File with a Standard Input or Output Stream” on page 47  
 “Define an Input Operator for a Class Type”  
 “Define an Output Operator for a Class Type” on page 53  
 “Correct Input Stream Errors” on page 55  
 “Format Stream Output” on page 23  
 “Define Your Own Format State Flags” on page 28  
 “Manipulate Strings with the `stringstream` Classes” on page 58  
 “Create Manipulators” on page 32

## 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"

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

    //...
}
```

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

### 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;
```

#### RELATED CONCEPTS

I/O Stream Classes

#### RELATED TASKS

“Combine Input and Output of Different Types” on page 46  
 “Use Predefined Streams” on page 20  
 “Receive Input from Standard Input” on page 36  
 “Display Output on Standard Output or Standard Error” on page 38  
 “Flush Output Streams with endl and flush” on page 40  
 “Parse Multiple Inputs” on page 42  
 “Open and Read from Files” on page 43  
 “Open and Write to Files” on page 46  
 “Associate a File with a Standard Input or Output Stream” on page 47  
 “Move through a file with filebuf Functions” on page 48  
 “Define an Output Operator for a Class Type”  
 “Correct Input Stream Errors” on page 55  
 “Format Stream Output” on page 23  
 “Define Your Own Format State Flags” on page 28  
 “Manipulate Strings with the stringstream Classes” on page 58  
 “Create Manipulators” on page 32

## 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
#ifdef NOSETW
        << setw(startingWidth)    // set width again
#endif
    << coord.Y
#ifdef NOSETW
        << setw(startingWidth)    // set width again
#endif
    << coord.Z;
    return aStream;
}
void main() {
    Coord_3D MyCoord(38.162168,1773.59,17293.12);
    cout << setw(17) << MyCoord << '\n'
        << setw(11) << MyCoord << endl;
}
```



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
```

#### RELATED CONCEPTS

I/O Stream Classes

#### RELATED TASKS

“Combine Input and Output of Different Types” on page 46

“Use Predefined Streams” on page 20

“Receive Input from Standard Input” on page 36

“Display Output on Standard Output or Standard Error” on page 38

“Flush Output Streams with `endl` and `flush`” on page 40

“Parse Multiple Inputs” on page 42

“Open and Read from Files” on page 43

“Open and Write to Files” on page 46

“Associate a File with a Standard Input or Output Stream” on page 47

“Move through a file with `filebuf` Functions” on page 48

“Define an Input Operator for a Class Type” on page 51

“Correct Input Stream Errors”

“Format Stream Output” on page 23

“Define Your Own Format State Flags” on page 28

“Manipulate Strings with the `stringstream` Classes” on page 58

“Create Manipulators” on page 32

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

This program loops indefinitely, given an input such as `ABC12`, because the erroneous input causes the error state to be set in the stream, but does not clear the

error state or advance the get pointer in the stream beyond the erroneous characters. Each time the input operator is called for an int (as in the while loop above), the same characters are read in.

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

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

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

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

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

#### RELATED CONCEPTS

I/O Stream Classes

#### RELATED TASKS

- “Combine Input and Output of Different Types” on page 46
- “Use Predefined Streams” on page 20
- “Receive Input from Standard Input” on page 36
- “Display Output on Standard Output or Standard Error” on page 38
- “Flush Output Streams with endl and flush” on page 40
- “Parse Multiple Inputs” on page 42
- “Open and Read from Files” on page 43
- “Open and Write to Files” on page 46
- “Associate a File with a Standard Input or Output Stream” on page 47
- “Move through a file with filebuf Functions” on page 48
- “Define an Input Operator for a Class Type” on page 51
- “Define an Output Operator for a Class Type” on page 53
- “Format Stream Output” on page 23
- “Define Your Own Format State Flags” on page 28
- “Manipulate Strings with the stringstream Classes” on page 58
- “Create Manipulators” on page 32

## Manipulate Strings with the stringstream Classes

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

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

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

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

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

```
// Using the stringstream classes to parse an argument list
#include <stringstream.h>
char* add(char*);
void main()
{
    cout << add("1 27 32.12 518") << endl;
}
char* add(char* addString)
{
    double value=0,sum=0;
    stringstream TwoWayStream;
    ostream OutputStream;
    TwoWayStream << addString << endl;
    for (;;)
    {
        TwoWayStream >> value;
        if (TwoWayStream) sum+=value;
        else break;
    }
    OutputStream << "The sum is: " << sum << "." << ends;
    return OutputStream.str();
}
```

This program produces the following output:

The sum is: 578.12.

## **RELATED CONCEPTS**

I/O Stream Classes

## **RELATED TASKS**

- “Combine Input and Output of Different Types” on page 46
- “Use Predefined Streams” on page 20
- “Receive Input from Standard Input” on page 36
- “Display Output on Standard Output or Standard Error” on page 38
- “Flush Output Streams with endl and flush” on page 40
- “Parse Multiple Inputs” on page 42
- “Open and Read from Files” on page 43
- “Open and Write to Files” on page 46
- “Associate a File with a Standard Input or Output Stream” on page 47
- “Move through a file with filebuf Functions” on page 48
- “Define an Input Operator for a Class Type” on page 51
- “Define an Output Operator for a Class Type” on page 53
- “Correct Input Stream Errors” on page 55
- “Format Stream Output” on page 23
- “Define Your Own Format State Flags” on page 28
- “Create Manipulators” on page 32