

# **TopSpeed® C++**

**For IBM® Personal Computers and Compatibles**

## **Class Library Guide**

**TopSpeed Corporation**

Copyright© 1990-1991, by TopSpeed Corporation. All rights reserved.

TopSpeed® is a registered trademark of TopSpeed Corporation.

Other brand or product names are trademarks or registered trademarks of their respective holders.

Printed in the United Kingdom.

10 9 8 7 6 5 4 3 2 1

# Contents

## CHAPTER 1

### **USING THE LIBRARY 8**

Who this manual is for .....	8
Using this manual .....	9
Using the classes .....	12

## CHAPTER 2

### **THE COMPLEX NUMBER CLASS 18**

Defining new types: the C++ solution .....	20
Summary of complex class members .....	22
Member functions and operators .....	24
Complex Class Directory .....	26
Public operators .....	26
Related global operators (friend operators) .....	27
Related global functions (friend functions) .....	27
Transcendental functions .....	28
Operators .....	28
Trigonometric functions .....	30

## CHAPTER 3

### **BCD CLASS 33**

<b>BCD Class directory 34</b>	<b>34</b>
Constants .....	34
Constructors .....	34
Public operators .....	34
Related global operators (friend operators) .....	34
Related global functions (friend functions) .....	35
<b>BCD Class Reference 36</b>	<b>36</b>
Constructors .....	36
friend functions .....	36
member functions .....	37

**CHAPTER 4****TASK AND PROCESS CLASSES 39**

Deriving a task from the abstract base class ..... 39

**Class directory 41**

Class task ..... 41

class semaphore ..... 41

class critical\_region ..... 42

class event ..... 42

**Class reference 43**

class task ..... 43

class semaphore ..... 44

class critical\_region ..... 45

class event ..... 45

**CHAPTER 5****WINDOW CLASSES 47****Class Directory 48**

class window ..... 48

class palettewindow: window ..... 50

**Class Reference 51**

class window ..... 51

class palettewindow: window ..... 54

**CHAPTER 6****STREAM CLASSES 56**

Weaknesses of C input-output ..... 56

The C++ solution ..... 57

**The stream class hierarchy 59**

The ios class and the streambuf ..... 59

**Elementary input-output 61**

The operators &lt;&lt; and &gt;&gt; ..... 61

Character (unformatted) reading and writing ..... 62

Moving upstream and downstream ..... 63

A simple example ..... 63

**More sophisticated input: exception handling 65**

The stream state ..... 65

A short note about encapsulating objects .....	66
Input control with stream state functions. ....	67
<b>Interactive input-output and flushing</b> .....	<b>69</b>
Automatic flushing: the tie function .....	69
<b>Format Control</b> .....	<b>71</b>
Default formats .....	71
The format state variable .....	71
Format functions .....	72
Field Widths .....	74
<b>Extending the stream classes to new types</b> .....	<b>78</b>
User defined stream operators .....	78
A footnote on assignable streams .....	82
<b>File streams</b> .....	<b>84</b>
Opening a file .....	84
Positioning within a stream .....	85
Binary input and output .....	86
<b>Incore formatting</b> .....	<b>88</b>
Storage allocation for incore formatting .....	88
<b>Defining manipulators</b> .....	<b>90</b>
Parameterized manipulators .....	91
From function to manipulator .....	92
Generic macros for types other than int and long .....	94
<b>Building your own streams</b> .....	<b>96</b>
Introduction .....	96
The streambuf class .....	96
The public and protected interface .....	97
The streambuf public interface .....	97
The protected interface: deriving new streambuf classes .....	102
Using streambufs in streams .....	105
Setbuf .....	105
Overflow .....	105
underflow .....	106
sync .....	106
Extending streams .....	106
Specializing istream or ostream .....	106
Extending state variables .....	106
Creating streams .....	108

<b>Conversion from 1.2</b>	<b>109</b>
Converting from streams to iostreams .....	109
streambuf internals .....	109
Incore formatting .....	109
Filebuf .....	111
Interactions with stdio .....	111
Assignment .....	112
char insertion operator .....	112
<b>Stream class directory</b>	<b>114</b>
header files .....	114
Inheritance structure .....	114
class ios -- base class .....	116
class streambuf -- base class .....	118
class istream: -- virtual public ios .....	119
class ostream: -- virtual public ios .....	120
class iostream: -- public istream, public ostream .....	121
class ostream_withassign: -- public ostream .....	122
class iostream_withassign: -- public iostream .....	122
class filebuf: -- public streambuf .....	122
class fstreambase: -- virtual public ios .....	123
class ifstream: -- public fstreambase, public istream .....	124
class ofstream: -- public fstreambase, public ostream .....	124
class fstream: -- public fstreambase, public iostream .....	124
class strstreambuf: -- public streambuf .....	125
class strstreambase: -- public virtual ios .....	125
class istrstream: -- public strstreambase, public istream .....	125
class ostrstream: -- public strstreambase, public ostream .....	126
class strstream: -- public strstreambase, public iostream .....	126
<b>Stream class reference section</b>	<b>127</b>
Conventions .....	127
class ios .....	127
class istream .....	135
class istream_withassign .....	136
ostream - formatted and unformatted output .....	141
class streambuf .....	145
strstreambuf - streambuf specialized to arrays .....	153
filebuf - streambuf specialized to files .....	155
stdiobuf - iostream specialized to stdio FILE .....	155

stringstream - iostream specialized to arrays ..... 155

## INDEX

**158**

# CHAPTER 1

## USING THE LIBRARY

The ANSI standard C language library is supplied with TopSpeed C++ and is fully link-compatible with it.

TopSpeed C++ also provides some extensions:

- A standard base class library. This implements ATT's release 2.00 stream and complex classes.
- A BCD class.
- A set of task and process-handling classes which are compatible with TopSpeed's other multi-tasking libraries.
- A set of text windowing classes which are compatible with TopSpeed's other window libraries.

These classes constitute a comprehensive toolkit for professional applications development.

This manual documents the standard base class library. The ANSI standard C library is the subject of a separate manual.

### Who this manual is for

---

This manual is a guide as well as a reference. It does not assume you are familiar with the classes it describes, though it will help if you have a nodding acquaintance with them.

The manual does assume you are familiar with the C++ language. Together with the TopSpeed C++ Tutorial it will serve as a complete introduction to the more standard classes which have grown up with C++, and a useful bridge to the more advanced classes supplied with the Rogue Wave library and with third party class libraries.

If you are already familiar with using classes from C++ libraries, you may still find it useful to read the next section which explains the structure of the manual, but you can skip the section after which introduces the classes themselves.

## Using this manual

---

### **Classes supplied with the library**

Five broad groups of classes are supplied with the standard library

- A complex number class
- A BCD class
- A set of task and process handling classes.
- A set of stream classes for I/O, incore data conversion and data manipulation.
- A set of text window classes and console I/O streams.

### **Selecting a library**

When using standard project files the correct C++ and C libraries will automatically be linked, matching operating system and memory model. No action from you is necessary.

When using customized project files containing #dmlink the C and C++ libraries must be added to the link list explicitly. The following line must be included in the project file to link the C++ library:

```
#pragma link(C%M%C%SLIB.LIB)
```

When using dynalink model the operating system macro must also be used:

```
#pragma link(%O%M%C%SLIB.LIB)
```

For details of linking C libraries explicitly see 'C Library Reference'.

### **Inline functions in iostream.hpp**

By default non-inline versions of library member functions in iostream.hpp are used. This both minimizes program size and maximizes compilation speed.

To enable small inline functions the macro `_SMALL_INLINE` must be defined in either the project file or in the program source file before the point of inclusion.

To enable the larger inline functions both the macros `_SMALL_INLINE` and `_BIG_INLINE` must be defined.

### **An object oriented manual**

A class library is different from a function library because it extends the data types available to the programmer as well as the functions.

You use a library class to create new objects; You use the manual to find out their properties. These are defined by their public functions and members. You might never want to go any further than declaring and using streams or, complex numbers; in this case you need only consult the manual pages that deal with public members.

### **Not just a load of old objects**

However, a well-designed class library lets you go further: it is designed so you can derive new classes. The structure aims to ensure this can be done cleanly and transparently, preserving the encapsulation of the library classes and guaranteeing portability.

The logical structure of the classes is therefore every bit as important as what they do. With this in mind, this manual has been designed to orient the user new to the C++ world, as well as simply listing the extra classes available. It prepares you to use the library classes — and also understand and extend them.

### **Finding your way around**

Because the C++ class library provides new data types as well as new functions, and because these new types are built in a logical and structured way, it is not helpful simply to list functions in alphabetical or lexical order.

The manual reflects the logical structure of the classes. The approach is close to that used in SmallTalk language manuals where it has become common to provide an ‘encyclopedia of classes’ listing the members of each class. In this manual each set of related classes is documented in a separate chapter.

Thus there is no single place you can go in the manual to find out how the + operator ‘works’ because it is defined differently for BCD and for complex data types. To find out how complex + works, look in the complex class chapter; to find out how BCD + works, look in the BCD chapter.

To help you find members quickly, we have supplied a ‘member index’ at the end of the manual. This contains a lexically-ordered list (beginning with the operators) of the members of all classes defined in the library. Where a function is overloaded you will find the names of all the classes which implement it with page references for their declaration and definition.

The member index also contains, for each member, a list of the classes which make up its formal parameters, also with page references.

### **What the class chapters contain**

Each class chapter is divided into three sections:

- A general introduction;

- A class directory;
- A formal reference section.

### **The introduction**

This explains how the classes fit together, with tips and pointers on their use. Read this if you are unfamiliar with the classes and want to find out how they work.

### **The class directory**

This is the Burke's Peerage of the class hierarchy. Read it for the pedigree and the formal declaration of the members and friends of each class, including the parameters they use and the values they yield. Use it also to survey all the members of a class at a glance.

### **The reference section**

Where the class directory supplies declarations, the reference pages define usage. Read it when you need to know exactly what each member does, what it doesn't do and how it can go wrong.

### **Public and protected interfaces**

Each list of class members is further divided up to distinguish between

- public members
- protected members

The public members are those you use if you just want to declare new objects and reference them, without deriving new classes of your own.

The protected members are those you might need if you are going to derive your own classes from the standard ones — either to customize the standards or to develop specialized classes from them.

### **Summary**

You can use this manual in one of two ways:

- You can browse it to find out what each class does and how.
- You can use it as a reference work, either by going straight to the reference or directory sections of the relevant class, or by referring first to the member index.

## **Naming conventions**

C++ can be a very ambiguous language with many overloaded identifiers. This makes naming conventions, more than usually important, even in examples.

The conventions adopted here are:

- Instances of most classes are designated by names which include the name of the class. Unless otherwise stated, `aStreambuf` and `thisStreambuf` refer to instances of class `streambuf`, `aBCD` to an instance of class `BCD`, and so on;
- Numeric magnitudes follow the FORTRAN convention: `i`, `j`, `k` etc stand for integers, `x`, `y`, `z` etc stand for doubles;
- Complex variables are called `xx`, `yy`, `zz` etc.
- Notwithstanding the above, where a descriptive name for a variable will aid clarity without ambiguity it may be used: for example in the complex constructor `polar(aMagnitude, anAngle)`.

When referring to functions within text we omit final brackets unless the reference is to the result of calling the function, rather than the function itself: `'printf'` rather than `'printf()'`, but `'aStream.rdbuf()'` to refer to the `streambuf` member of a `stream` class instance, which is normally obtained by invoking the `rdbuf` member function.

## **Using the classes**

---

The TopSpeed C++ philosophy is to provide state-of-the-art tools for the C++ developer, combined with the highest level of standardization possible at this stage of C++ development.

C++'s ability to define new types brings both benefits and pitfalls. It takes the creation of data types out of the hands of the language designer and puts it in the hands of the application programmer. It turns every programmer into a language developer.

### **Development versus standardization**

This brings the danger that data types and their treatment will proliferate wildly, with all the attendant risks that code will be non-portable and non-modular. These risks are lessened when user-defined classes become standardized. But standardized classes are less flexible. Programmers always have to strike a balance between flexibility and standardization.

As a young language C++ is still developing, and its associated libraries are the fastest area of change. Many powerful classes, for example collection

classes, which define objects such as sets and dictionaries, are still in a state of flux and vary widely from implementation to implementation because of the differing philosophies of their designers.

This is a fundamental difference between C++ and SmallTalk, which provides a large and carefully-constructed library of classes that is so closely identified with the language that in effect it is the language.

Nevertheless some C++ classes, for example streams, have been with it since its early days and nearly every C++ implementation will provide them in one form or another.

### **First steps to a standard**

As the C++ world develops, its more tried and tested classes move towards standardization. The programmer can begin to work with classes and members which are in very widespread and common use with confidence that they will be supported on a wide range of platforms and over a considerable time-span.

The ATT class library probably represents the closest there is to a de-facto standard, particularly in the area of streams and input-output. ATT's release 2.0 classes greatly extend the early, somewhat limited C++ release 1.2 stream classes, offering for the first time a genuinely comprehensive set of base classes and members from which the developer may confidently branch out.

### **Using class objects**

The simplest use of a standard class is to use it to create objects. In the library you will find classes such as the complex number class and the istream and ostream classes, which it is intended you should use just like a new type.

Thus, to take the simplest case, if you want to create a complex number you will simply write something like

```
complex c;
```

in your program.

In such a case you need to know

- how to create new objects.
- which operators can be used with them
- which functions can be used with them
- what these operators and functions actually do, and what restrictions there are on their use.

- what error conditions can arise from using them and how these conditions are handled.

Each of these questions is dealt with in a separate section in the chapter dealing with the class concerned.

### **Creating new objects**

To find out how to create objects you should look to the associated constructors. Remember that constructors in C++ are often invoked when you don't expect; for example implicit type conversion will invoke a constructor, as will an initialized declaration.

### **Using new objects**

When you are trying to find out what a class can do you have to look in two places: at its member functions and operators, and at the overloaded friend functions and operators associated with it.

In the complex class, for example, we find the member operator += defined as follows:

```
complex& complex::operator +=(complex&);
```

but the + operator is defined as a set of friend functions overloading the normal + operator:

```
complex& operator +(complex&, complex&);  
complex& operator +(double&, complex&);  
complex& operator +(complex&, double&);
```

This is sometimes confusing when you are looking for an operator because the syntax which invokes it is the same in both cases. For a function the distinction is clearer.

The difference is subtle but not unimportant. A member operator is tied to the class more tightly; only objects from the class can invoke it. An overloaded function can be invoked as a result of an implicit type conversion. Moreover the user can define extra overloaded functions but may not define new members.

You can find some examples and more detail in the introduction to the complex class chapter. At this point you should just note that you must study both the member functions of a class and any functions which take it as a parameter, in order to find out what it does.

### **Restrictions and errors**

Some library classes are abstract, containing pure virtual functions, or may contain un-implemented constructors. An attempt to create an object of these types will cause a compile or run-time error.

## **Public, private and protected**

TopSpeed's C++ library makes full and proper use of C++'s data encapsulation and data hiding features. When you use a class, you don't have access to its representation or to the implementation of its members, and in principle you don't have to know what they are: you only need to know what they do.

The library classes conceal internal representation by distinguishing between public and private class members; the representation is effected through the private members of the class, and you only use the public ones in your applications.

## **Creating new classes**

The library classes are more than just a set of new types. They also form a set of base classes for creating your own types.

You can create custom classes by inheriting from the library classes. Within your derived classes you can use the full power of the library classes but you can also modify or change their operation.

This more sophisticated use requires more knowledge of the way the classes work internally. If you modify a base class member in a derived class, you have to be aware of the way this might affect other base class members which use the member you have adapted.

The `streambuf` class, for example, which is responsible for fetching characters for the stream classes to use, provides a protected virtual function called `underflow` which is invoked whenever it is asked for input and doesn't have any. `Underflow` is itself called by several public `streambuf` functions, for example the function `sgetc` which fetches a single character from the input stream. When you derive a class from `streambuf` you supply your own `underflow` function. It is important to know that the new `underflow` will be invoked whenever you call `sgetc`, or you may get side-effects you do not expect.

When you want to modify or build on a class, you need more access to its representation. Some of the classes therefore provide protected members which are for use in building derived classes.

This manual separates public from protected members for precisely this reason: they serve different purposes. For classes like the `streambuf` class mentioned above, which is intended to be used widely for deriving other classes, the manual distinguishes between the protected interface, used by the derived classes, and the public interface, used by the rest of the application.

## **Encapsulation and implementation independence**

Although you have access to a base class's protected members your derived classes cannot access its private members. This is as it should be; it ensures that if the implementation or representation of the base classes is changed, your derived classes will still work.

In release 1.2 of ATT's stream classes the logical design was incomplete in certain respects, and certain applications found it necessary to use features of the streambuf class's internal representation. Release 2 corrects these weaknesses and it is no longer necessary to do this.

However earlier applications using the streambuf class may now have to be rewritten, because the internal implementation of the class has changed. This shows how important it is to design base classes which are genuinely encapsulated and do not force the user to use 'internal' class features. It also shows how important it is not to use implementation-dependent or private features of the base classes.

## **Redefining class members**

On occasions you may wish to redefine or re-implement members of a library class. The best way to do this is to create a customized class of your own in which you redefine selected member functions.

Suppose, for example, you want to redefine the complex class constructor so that it will not accept magnitudes greater than a certain quantity. Class complex has a member function abs which returns the norm  $|zz|$  of a complex number zz. You can define a new class which will invoke the standard constructor but carry out additional checks of its own:

```
class mycomplex: complex
{
    mycomplex(
        double realPart,
        double imaginaryPart):
        complex(realPart, imaginaryPart)
    {
        if (abs() > MAXCOMP)
            error("complex too large");
    }
}
```

This is the preferred method for redefining the member functions of library classes. Unfortunately C++'s encapsulation is not perfect, and you may find you can redefine a class member by simply compiling and linking in your own definition of a particular member. You will also notice that inline functions are defined in the include files (since the inline function source code must be available to the compiler when it comes across a class to it), giving you the possibility to change inline functions by rewriting header files.

We do not recommend this approach and cannot guarantee to support programs which attempt redefinition in this way. One of the purposes of encapsulation is to prevent applications changing the internal implementation of a class. This brings many advantages:

- It modularizes the debugging process. Once a class is debugged, if you are not allowed to change it, you cannot reintroduce errors in it.
- It makes code more portable because it guarantees it can be shared. If you cannot change a class definition, then you can be sure that two users of the class are working with the same definition
- It facilitates revisions and improvements to class libraries, since library suppliers can update the implementation without changing the external form of the library.

Finally, class implementations with a high degree of interconnectedness may rely on side effects. In this situation re-defining a member may produce unexpected results.

### **Summary**

In summary:

- Create objects using constructors, but bear in mind that constructors may be invoked as implicit type convertors;
- Use the objects you have created through the public class members and associated overloaded operators and functions.
- Build or customize classes using a combination of protected and public class members. But avoid implementation dependent features when you derive new classes.

## CHAPTER 2

# THE COMPLEX NUMBER CLASS

C++ has no built-in complex type. The class `complex` is a user-defined implementation of this type which serves as a useful introduction to C++'s type definition features.

We can understand the class best by making a list of the requirements which a new arithmetic type ought to satisfy (see Chapter 1 'naming conventions' for the identifiers used)

- The new type should be intuitively reasonable to use: it should implement the same operators as the built-in arithmetic types, but not where they have no meaning for the new type. Thus `xx + yy` is a perfectly reasonable complex expression and should be implemented by the new type, whereas `xx & yy` is not really useful or reasonable and should not be implemented.
- Any type-specific operators — unique to the new type — should be supplied. For example the functions `imag(zz)` and `real(zz)` return the imaginary and real parts of a `zz` respectively.
- The implementation should be efficient. The user should pay no run-time penalty for using the new type instead of programming in longhand.
- The new type should be usable in mixed expressions — in combination with other arithmetic types such as `double`, `float`, `int`. Conversion rules should be applied which conform to general C++ conventions. For example in the expression `x + zz`, `x` should be converted to a complex number, and the result should be complex.
- The library should offer the full range of arithmetic and mathematic functions, such as `cos`, `sin` and so on, which extend the existing library to the new type.
- The complex class should be encapsulated: its implementation should be concealed; and the user should not be able to introduce errors by tampering with the internal working of the class.
- It should be extendable within reasonable limits: the user should be able to add to its definition without interfering with its internal workings.

## **A note on inheritance**

There is a further requirement which we might term inheritability, which is adhered to by a language such as SmallTalk in which all types are placed on a logical hierarchy. For example types which implement the member `<` inherit from a common abstract class `magnitude`. The advantage of this is that any members implemented for the `magnitude` class are instantly available for any new type which inherits from `magnitude`.

If C++ provided such a structure, for example, there would be no need for each new class to implement members like `+=`, and `-=`. Provided the new type inherited from a class which defined `+=` in terms of the `+` and `=` operators, it would automatically implement `+=`.

This procedure is not usually followed in C++ because the language was designed as a natural progression from C and has to accept the existing, non-hierarchically-organized arithmetic types.

In consequence the implementation of the complex type makes less use of data abstraction than is possible, and — the most important practical consequence — every operation that might be needed by the new type has to be implemented by the complex class. There is no code re-use.

## **Borderline cases**

At first sight an expression such as `zz < yy` falls within the requirement that intuitively reasonable operators should be available. It can be interpreted as

```
abs(zz) < abs(yy)
```

which we could read as “zz is closer to 0 than yy is.” However, the `<` operator so defined does not quite behave as we might expect. We can see this by asking whether it squares with the intuitively reasonable notion that if `xx < yy` is false and `yy < xx` is also false, then

```
xx == yy
```

For all other arithmetic types this is true. But for complex types it would not be, if we used the above definition: for example if `xx` is the number `1 + i` and `yy` the number `-1 - i` then neither `xx < yy` nor `yy < xx` holds. But neither does `xx == yy`. The approach taken in this library is cautious. Operators such as `<` are not implemented, but the user can extend the class to include them by overloading existing operators.

Within these constraints the library class `complex` is thorough and efficient.

## Defining new types: the C++ solution

---

### Options in defining new types

There are a variety of ways a new type can be created. Broadly speaking the choices available at the current stage of language development are:

- Extend the language definition
- Use a macro facility such as C's preprocessor
- Use object-oriented languages to create a user defined type.

C++ is designed to facilitate the third option.

The problem with extending the language is that the variety of conceivable new types is infinite; also, new types are often quite specialized (for example matrix types) and can create a language which is heavily loaded down with features only used by a small cross-section of users. The result is a heavy penalty in compiler speed and memory usage for which most users get no benefit.

The second approach abandons semantic controls. All the safeguards provided by strict type-checking are switched off, and debugging becomes much harder. In particular, it becomes hard to adopt quality-assurance procedures whose aim is error-prevention, rather than error-correction.

### The C++ option: a new class

The C++ approach is to create a new class, class complex, equipped with

- Constructors so that complex objects can be created;
- Member operators such as +, - which re-implement the standard arithmetic operations for the new type;
- Friend library functions such as cos and sin which re-implement standard arithmetic library functions for the new type;
- Type-specific friend functions such as conj (the conjugate of a number) which are specific to the new type.

### How the new class solves other problems

Because of C++'s handling of constructors, overloading and implicit type conversion, many of our desired wish-list features come out in the wash.

However the mechanisms involved are quite subtle, and not always clear to the first-time C++ programmer.

As long as you confine yourself to using the class as it stands few problems are likely. If, however, you want to redefine complex class members or friend operators you should be aware of what goes on because there are a number of pitfalls to be avoided.

Let us consider type conversion as an example.

### **Friend operators as type converters**

The complex class deals with the expression

```
zz + x
```

by defining the operator `+` as a friend operator of class `complex`. The TopSpeed C++ library overloads the standard operators heavily so as to avoid inefficient type conversions. There are in fact three complex `+` operators:

```
complex operator + (complex&, complex&);
complex operator + (double, complex&);
complex operator + (complex&, double);
```

In the example above, the third version of `+` is selected, because `x` is a `double`. We could have defined just one:

```
complex operator + (complex&, complex&);
```

If we did this, the expression `zz + x` would still be correctly handled. This is because the constructor `complex(double)` can create anonymous complex objects: it serves, in fact, as an implicit type convertor for the class. If the compiler finds a `double` in a context where a `complex` is required, it will see if there is a way to build one from the `double`. The constructor provides just such a means, so it would be applied to convert it. `x` would be converted to a `complex` using this constructor, and then added to `c`.

This would result in a less efficient calculation because a temporary complex variable would be created; the TopSpeed library simply adds `x` to the real part of `c`.

Note, however, that no further overloading is needed — for example for integers — because C++'s built-in type-promotion rules ensure that expressions such as

```
zz + i
```

are efficiently evaluated by promoting `i` to a `double` in order to decide which of the overloaded operators to choose.

### **Member functions restrict type conversion**

Implicit type conversions that go on behind the programmer's back can be one of the most dangerous features of C++.

This is one of the reasons the assignment operators `=`, `+=` and so on are defined as member operators, not as friend operators.

The result is that the left-hand operand in an assignment cannot be unexpectedly converted. If the assignment operator had been defined as a friend function

```
friend complex& operator =(complex&, complex&);
```

then on many compilers the following monstrosity would compile and execute:

```
x += zz; //wrong!
```

Version 2.1 of C++ explicitly prohibits this with a rule that says an anonymous temporary cannot be used as a reference, probably because such constructions used to be possible. This was because the complex class possesses a constructor

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

This is invoked in declarations such as

```
complex zz = 1;
```

Here the constructor `complex(1, 0)` is invoked because the second parameter defaults to 0. Unfortunately, C++ rules say that any constructor can be used as a type converter; so `x` would be converted to a temporary complex, assigned the value 0, added to `zz` and then thrown away.

There is nevertheless a choice between implementing an operator as a member or as a friend. Opting for a member operator is the most efficient and minimizes ambiguity.

Let us now look at the full range of facilities which the complex class offers.

## Summary of complex class members

---

In what follows we assume some knowledge of basic stream class operators. Refer to chapter : “Stream classes” for further details.

### Representation and initialization

The complex class provides constructors which permit complex class objects to be created in an intuitively obvious way.

```
complex zz(3,-5);
```

will declare `zz` to be complex with the value  $3 - 5i$ . The first value of the pair is thus taken as the real part of the Cartesian representation of a complex number and the second as the imaginary part. The constructor converts the arguments to type `double`. A complex variable can be initialized to a real value by using the constructor with only one argument. For example:

```
complex zz = complex(1);
```

will set up zz as a complex variable initialized to (1,0).

A complex variable which is not explicitly initialized is set to (0,0). For example:

```
complex orig;
```

is equivalent to:

```
complex orig = complex(0,0);
```

The advantage of using constructors instead of a preprocessor statement is immediately apparent. Because a constructor can define anonymous types all normal variants become available. For example:

```
complex xx(3,-5);
complex*yy = new complex(0, 2);
complex zz = complex(1,2) + complex(2,-1)
complex xx = 123;
```

The last declaration sets xx to be the complex number  $123 + 0i$ . No conversion of a complex into a double is defined, so

```
double x = complex(1,0);
```

is illegal and will cause a compile time error. However as pointed out previously, the constructor `complex(double, double = 0)` serves as an implicit type convertor from doubles to complexes.

Naturally a complex variable can also be initialized by a complex expression. For example:

```
complex zz(-0.5000000e+02,0.8660254e+02);
complex yy = xx+log(zz);
```

## **Constructed types**

It is also possible to declare arrays of complex numbers. For example:

```
complex carray[30];
```

sets up an array of 30 complex numbers, all initialized to (0,0). Using the above declarations:

```
complex carr[] = {c,d,carray[2],complex(1,2)};
```

sets up a complex array carr[] of four complex elements and initializes it with the members of the list.

## **Encapsulated internal representation**

While both the initialization functions and the stream i/o functions accept the Cartesian representation of a complex, the internal representation is inaccessible and in principle unknown to a user.

Therefore a struct style initialization cannot be used. For example:

```
complex cwrong[] = {1, 3, 4.1, -24};
```

is illegal, because it makes unwarranted assumptions about the representation of complex numbers.

## Member functions and operators

---

### Basic operators

The complex class provides the basic arithmetic operators +, /, \*, and -, the assignment operators =, +=, -=, \*=, and /=, and the comparison operators == and !=.

The operators +=, -=, \*=, and /= do not produce a value that can be used in an expression. Thus the following examples will cause compile time errors:

```
if ((yy+=2)==0)...; //wrong
zz = yy *= zz; //wrong
```

### Mathematical functions

The standard transcendental functions sqrt, exp, log, sin, cos, sinh, cosh, and pow are extended to complex numbers.

Other trigonometric and hyperbolic functions, for example tan and tanh, can be written by the user using overloaded function names.

These function names are overloaded. When called, the function to be invoked will be chosen based on the argument type. For example, log(1) will invoke the real log, and log(complex(1)) will invoke the complex log. In each case the integer 1 is converted to the real value 1.0.

The type-specific functions which one would expect are provided. real and imag return the real and imaginary parts of a complex number respectively, while conj provides the conjugate complex of a complex number

**norm(c)** returns the square of the magnitude of c. It is faster than abs(c), but more likely to cause an overflow error. It is intended for comparing magnitudes.

Polar coordinates can be used. The function polar creates a complex given its polar representation, and abs and arg return a complex number's polar magnitude and angle. The function norm returns the square of a complex magnitude.

### Input and output

Input and output operators >> and << are also provided. << prints a complex as (real,imaginary) and >> reads it in the same format.

They are declared as stand-alone functions using the facility for overloading function operators:

```
ostream& operator<<(ostream&, complex);
istream& operator>>(istream&, complex&);
```

When `zz` is a complex variable `cin >> zz` reads a pair of real numbers from the standard input stream `cin` into `zz`. The first number of the pair is interpreted as the real part of the Cartesian representation of a complex number and the second as the imaginary part. No delimiters are necessary although the following format is accepted:

```
(1.234e123, 1.234e123)
```

The expression `cout << zz` writes `zz` to the standard output stream `cout`, enclosed by brackets and delimited by a comma. Thus for example:

```
void copy(istream& from, ostream& to)
{
    complex zz;
    while (from >> zz)
        to << zz;
}
```

reads a stream of complex numbers, like

```
3 4
2 -1
6 1
```

and writes them like

```
(3, 4)
(2,-1)
(6, 1)
```

The precise output format will depend on the conversion settings for floating point.

A single real number, for example `10e-6` or `(123)`, will be interpreted as a complex with 0 as the imaginary part by operator `>>`.

A user who does not like the standard implementation of `<<` and `>>` can provide alternate versions. The example below shows how user-defined output functions can modify the operation of the complex class using the standard public access member functions `real` and `imag`:

```
complex zz = complex(3.4,5);
cout
    << real(zz) << "+"
    << imag(zz) << "*i";
```

will print `3.400000e00+5.000000e00*i`.

This can be extended to provide a polar co-ordinate representation.. For example:

```

complex zz = polar(sqrt(2), PI/4); // = 1 + i
double magn = abs(zz); // = sqrt(2)
double angl = arg(zz); // = PI/4
cout
    << "(m=" << magn
    << ", a=" << angl
    << ")";

```

If input and output functions for the polar representation of complex numbers are needed they can be written by the user.

### **Error handling**

Errors in complex class member functions fall into two categories:

- Errors arising in standard library math functions.
- Errors arising in the code of a member function.

### **Errors in standard math functions**

The default behavior of the function concerned will occur. This may be modified by an error handling function assigned to the `matherr` function pointer.

### **Errors in member functions**

The default behavior on the generation of a floating point exception outside of the standard math library is to terminate the process. This behavior may be modified by installing a handler via the function `signal`.

## **Complex Class Directory**

Declarations and definitions for class `complex` are found in `complex.hpp`

### **Constructors**

```

complex(double _re_val,          double _im_val=0);
complex();

```

## **Public operators**

### **Assignment**

```

complex& operator += (complex&);
complex& operator += (double);
complex& operator -= (complex&);
complex& operator -= (double);
complex& operator *= (complex&);
complex& operator *= (double);
complex& operator /= (complex&);
complex& operator /= (double);

```

## Unary

```
complex operator + ();  
complex operator - ();
```

## Public function members

Class `complex` has no public function members.

## Related global operators (friend operators)

---

### Arithmetic

```
complex operator + (complex&, complex&);  
complex operator + (double, complex&);  
complex operator + (complex&, double);  
complex operator - (complex&, complex&);  
complex operator - (double, complex&);  
complex operator - (complex&, double);  
complex operator * (complex&, complex&);  
complex operator * (double, complex&);  
complex operator * (complex&, double);  
complex operator / (complex&, complex&);  
complex operator / (double, complex&);  
complex operator / (complex&, double);
```

### Comparison

```
int operator == (complex&, complex&);  
int operator != (complex&, complex&);
```

## Related global functions (friend functions)

---

### Type-specific functions

#### Decomposition

```
double real(complex&);  
double imag(complex&);
```

#### Polar

```
double arg(complex&);  
complex polar(double, double = 0.0);
```

#### Metric

```
double norm(complex&);  
double abs(complex&);
```

#### Other

```
complex conj(complex&);
```

## Transcendental functions

---

### Trigonometric

```
complex cos(complex&);
complex sin(complex&);
complex tan(complex&);
```

### Inverse trigonometric

```
complex acos(complex&);
complex asin(complex&);
complex atan(complex&);
```

### Hyperbolic

```
complex cosh(complex&);
complex sinh(complex&);
complex tanh(complex&);
```

### Other

```
complex exp(complex&);
complex log(complex&);
complex log10(complex&);
complex pow(complex&, double exponent);
complex pow(double, complex& exponent);
complex pow(complex&, complex& exponent);
complex sqrt(complex&); Complex Class Reference
```

### Constructors

```
complex zz(x, y);
```

Creates a complex number whose real part is x and whose imaginary part is y.

```
complex zz(x);
```

Creates a complex number whose real part is x and whose imaginary part is 0.

```
complex();
```

Creates a complex number whose real and imaginary part are both 0.

## Operators

---

The basic assignment, arithmetic and comparison operators are overloaded for complex numbers. They have their conventional precedence and associativity.

### Assignment operators

```
xx += yy;
```

Complex number xx is assigned the value of the arithmetic sum of itself and complex number yy.

```
xx -= yy;
```

Complex number `xx` is assigned the value of the arithmetic difference of itself, and complex number `yy`.

```
xx *= yy;
```

Complex number `xx` is assigned the value of the arithmetic product of itself and complex number `yy`.

```
xx /= yy;
```

Complex number `xx` is assigned the value of the arithmetic quotient of itself and complex number `yy`.

**Caution: The assignment operators do not produce a result, so that assignment operators cannot be stacked. The following construction is therefore invalid:**

```
complex xx, yy, zz;
xx = ( yy += zz ); //wrong: see above
```

### **Unary arithmetic operators**

```
zz = -yy;
```

Yields  $0 - yy$ ;

```
zz = +yy;
```

Yields  $0 + yy$ .

### **Binary arithmetic operators**

```
zz = xx + yy;
```

Returns a complex which is the arithmetic sum of complex numbers `xx` and `yy`.

```
zz = xx - yy;
```

Returns a complex which is the arithmetic difference of complex numbers `xx` and `yy`.

```
zz = xx * yy;
```

Returns a complex which is the arithmetic product of complex numbers `xx` and `yy`.

```
zz = xx / yy;
```

Returns a complex which is the arithmetic quotient of complex numbers `xx` and `yy`.

### **Comparison operators**

```
x == yy;
```

Returns non-zero if complex number `xx` is equal to complex number `yy`; returns 0 otherwise.

```
xx != yy;
```

Returns non-zero if complex number `xx` is not equal to complex number `yy`; returns 0 otherwise.

The following math functions are overloaded by the complex library so that they will also accept complex arguments:

### **Type-specific functions**

The following type-specific functions are defined for class `complex`, taking complex arguments and returning real results:

```
x = real(zz);  
Returns the real part of zz.  
x = imag(zz);
```

Returns the imaginary part of `zz`.

```
zz = polar(aMagnitude, anAngle);
```

Creates a complex, given a pair of polar coordinates, measured in radians in the range `+p` to `-p`.

```
x = arg(zz);
```

Returns the angle of `zz`, measured in radians in the range `-p` to `+p`.

```
x = norm(zz);
```

Returns the square of the magnitude of `zz`. It is faster than `abs`, but more likely to cause an overflow error. It is intended for comparison of magnitudes.

```
x = abs(zz);
```

Returns the absolute value or magnitude of `zz`.

```
x = conj(zz);
```

Returns the conjugation of `x`. That is, if `zz` is `(re, im)`, then `conj(x)` is `(re, -im)`.

## **Trigonometric functions**

The following trigonometric functions, returning complex results, are defined for complex arguments:

```
yy = sin(zz);
```

Returns the sine of `zz`.

```
yy = cos(zz);
```

Returns the cosine of `zz`.

```
yy = tan(zz);
```

Returns the tangent of `zz`.

### **Inverse trigonometric**

```
yy = acos(zz);
```

Returns the arc cosine of zz.

```
yy = asin(zz);
```

Returns the arc sine of zz.

```
yy = atan(zz);
```

Returns the arc tangent of zz.

### **Hyperbolic**

```
yy = sinh(xx)
```

Returns the hyperbolic sine of xx.

```
yy = cosh(xx)
```

Returns the hyperbolic cosine of xx.

```
yy = tanh(xx);
```

Returns the hyperbolic tangent of xx.

### **Other**

The following math functions are similarly overloaded by the complex library, accepting complex arguments and returning double arguments.

```
xx = exp(zz)
```

Returns e to the power zz, e being 2.718281828...

```
xx = log(zz)
```

Returns the natural logarithm of zz (to the base e). log(0) causes an error.

```
xx = pow(zz, y)  
xx = pow(y, zz)  
xx = pow(xx, zz)
```

Returns the first argument raised to the power of the second argument.

The following function is overloaded to take a complex argument and produce a complex result:

```
zz = sqrt(xx)
```

Returns the square root of zz, contained in the first or fourth quadrants of the complex plane. Note that sqrt(x) for doubles is not overloaded to produce a complex result, so that if you try and extract the square root of a negative real number you will still get an error.

Assigning the result to a complex makes no difference because C++'s overloading rules will not invoke the complex version of sqrt. The statement

```
zz = sqrt(x); // wrong: uses double sqrt
```

would call the double version of sqrt and convert the result to a complex (by assigning it to the real part of zz). To achieve the result intended above, write

```
zz= sqrt(complex(x)); //OK: uses complex sqrt
```

## **CHAPTER 3**

### **BCD CLASS**

For efficiency all arithmetical operations are carried out using real numbers, the bcd number being converted before and after the operation. This allows bcd, real and integral types to be mixed freely in expressions, but does not guarantee the most commonly desired attribute of bcd numbers - complete precision for large decimal numbers. Any over or underflow generated may cause precision to be lost, therefore the most useful range for integers in this implementation is -10e16 to 10e16.

When representing real numbers, the number of digits carried after the decimal point may be specified when constructing a bcd object with a real number. The default value, `_BcdMaxDecimals`, provides the maximum precision, about 17 digits.

## ***BCD Class directory***

Declarations and definitions for class bcd are found in bcd.hpp

### **Constants**

---

Class BCD defines one constant, `_BcdMaxDecimals`, which defines the maximum number of significant digits a BCD number may have and is currently set to 100, which represents the maximum.

### **Constructors**

---

```
bcd();  
bcd(int);  
bcd(unsigned);  
bcd(long);  
bcd(unsigned long);  
bcd(double, int = _BcdMaxDecimals);  
bcd(long double, int = _BcdMaxDecimals);
```

### **Public operators**

---

#### **Member operators**

##### **Assignment**

```
bcd& operator+=(bcd&);  
bcd& operator+=(long double);  
bcd& operator-=(bcd&);  
bcd& operator-=(long double);  
bcd& operator*=(bcd&);  
bcd& operator*=(long double);  
bcd& operator/=(bcd&);  
bcd& operator/=(long double);
```

##### **Unary**

```
bcd operator+();  
bcd operator-();
```

### **Related global operators (friend operators)**

---

### **Arithmetic**

```
friend bcd operator+(bcd&, bcd&);
friend bcd operator+(long double, bcd&);
friend bcd operator+(bcd&, long double);
friend bcd operator-(bcd&, bcd&);
friend bcd operator-(long double, bcd&);
friend bcd operator-(bcd&, long double);
friend bcd operator*(bcd&, bcd&);
friend bcd operator*(long double, bcd&);
friend bcd operator*(bcd&, long double);
friend bcd operator/(bcd&, bcd&);
friend bcd operator/(long double, bcd&);
friend bcd operator/(bcd&, long double);
```

### **Comparison**

```
friend int operator==(bcd&, bcd&);
friend int operator!=(bcd&, bcd&);
friend int operator>=(bcd&, bcd&);
friend int operator<=(bcd&, bcd&);
friend int operator> (bcd&, bcd&);
friend int operator< (bcd&, bcd&);
```

## **Related global functions (friend functions)**

---

### **Type conversion**

```
friend long double real(bcd&);
```

### **Metric**

```
friend bcd abs(bcd&);
```

### **Transcendental functions**

```
friend bcd acos(bcd&);
friend bcd asin(bcd&);
friend bcd atan(bcd&);
friend bcd cos(bcd&);
friend bcd cosh(bcd&);
friend bcd exp(bcd&);
friend bcd log(bcd&);
friend bcd log10(bcd&);
friend bcd pow(bcd&, bcd& exponent);
friend bcd sin(bcd&);
friend bcd sinh(bcd&);
friend bcd sqrt(bcd&);
friend bcd tan(bcd&);
friend bcd tanh(bcd&);
```

# ***BCD Class Reference***

## **Constructors**

---

Constructors are provided to initialize a bcd object with both integral and real types.

```
bcd(int x=0);
```

bcd() may be used, initializing the object to zero.

```
bcd(unsigned x);  
bcd(long x);  
bcd(unsigned long x);
```

A bcd object will be initialized with the integral value specified.

```
bcd(double x, int decimals);  
bcd(long double x, int decimals);
```

A bcd object will be initialized with the real value specified. Decimal digit precision will be set to the default value `_BcdMaxDecimals` unless specified.

## **friend functions**

---

```
long double real(bcd&);
```

Returns the a bcd value converted to real. In the case of bcd values greater than 10e17 accuracy is not guaranteed.

```
bcd abs(bcd&);  
Returns the absolute value of the bcd object.  
bcd acos(bcd&);  
bcd asin(bcd&);  
bcd atan(bcd&);  
bcd cos(bcd&);  
bcd cosh(bcd&);  
bcd exp(bcd&);  
bcd log(bcd&);  
bcd log10(bcd&);  
bcd pow(bcd& base, bcd& expon);  
bcd sin(bcd&);  
bcd sinh(bcd&);
```

The above functions convert the bcd value to a real number, call the appropriate math function and return that value as a bcd object. Error handling is as defined for the ANSI math library. (See C Library Reference).

## **Binary Operators**

```

bcd operator+(bcd&, bcd&);
bcd operator+(long double, bcd&);
bcd operator+(bcd&, long double);
bcd operator-(bcd&, bcd&);
bcd operator-(long double, bcd&);
bcd operator-(bcd&, long double);
bcd operator*(bcd&, bcd&);
bcd operator*(long double, bcd&);
bcd operator*(bcd&, long double);
bcd operator/(bcd&, bcd&);
bcd operator/(long double, bcd&);
bcd operator/(bcd&, long double);

```

The above binary operators provide an arithmetic capability for combinations of both real and bcd numbers. Errors such as overflows will be handled by the default floating point exception handling mechanism (See C Library Reference).

Operations involving large bcd numbers, multiplication or division may cause loss of precision.

## **Relational Operators**

```

int operator==(bcd&, bcd&);
int operator!=(bcd&, bcd&);
int operator>=(bcd&, bcd&);
int operator<=(bcd&, bcd&);
int operator>(bcd&, bcd&);
int operator<(bcd&, bcd&);

```

Since the bcd number is a scalar type the above set of relational operators is provided. All functions return non-zero if the condition is true, zero if false.

## **Stream Operators**

```

ostream& operator<<(ostream&, bcd&);
istream& operator>>(istream&, bcd&);

```

## **member functions**

---

### **binary operators**

```

bcd& operator+=(bcd&);
bcd& operator+=(long double);
bcd& operator-=(bcd&);
bcd& operator-=(long double);
bcd& operator*=(bcd&);
bcd& operator*=(long double);
bcd& operator/=(bcd&);
bcd& operator/=(long double);
bcd operator+();
bcd operator-();

```

The above unary and complex assignment operators extend the arithmetic capability for bcd numbers. Errors such as overflows will be handled by the

default floating point exception handling mechanism (See C Library Reference).

Operations involving large bcd numbers, multiplication or division may cause loss of precision.

# **CHAPTER 4**

## ***TASK AND PROCESS CLASSES***

The class task provides an abstract base class for the user to derive a class associated with a thread of execution. The underlying mechanism used for threads is the JPI process module, and task objects are compatible with threads created by the C, Pascal and Modula-2 JPI process modules. See C Library Reference.

The semaphore class is an implementation of a simple semaphore that may be unconditionally set or cleared, awaited or requested.

The critical region class may be used to protect a non-re-entrant region of code. For global process locking see Lock and Unlock in C Library Reference.

The event class provides the same facilities as a signal in the JPI process module. An object of this class may be used for inter-thread communication.

### **Deriving a task from the abstract base class**

---

There is a fundamental problem with implementing a thread in a preemptive multi-tasking environment. It is not possible to kill a thread externally and to guarantee that semaphores held by that thread will not cause a later deadlock. Therefore the only practical way for a thread to end is for it to terminate itself, possibly in response to a signal. For this reason a task object with dynamic or automatic storage will not necessarily have the same lifetime as the thread of execution associated with it. The thread procedure in a task object is declared to be static (i.e. with no this pointer and therefore no access to the task object) and must be defined in the derived class. Communication between the object and its associated execution thread is achieved via private static semaphores.

The first stage in creating a task object is to derive a class:

```

class task1 : public task {
public:
    task1();

protected:
    void (*proc_adr())();

private :
    static void thread_proc();
};

```

A constructor that calls the protected member function `init()` must be defined.

The first parameter is the address of the thread procedure which may either be the result of the function `proc_adr()` or the address of a friend function.

The second parameter is the desired stack size for the thread - a minimum of 2000 bytes is recommended.

The third parameter is the thread priority. A default value of 1 is used if this parameter is omitted. Threads of higher priority will receive more time slices than those of a lower priority. A thread may change its priority by calling `priority` by calling `set_priority()`.

The third parameter is the start flag. A non-zero value will cause the thread to begin execution when the object is constructed. If this parameter is omitted or is zero the thread will not begin execution until the member function `start()` is called.

```

task1::task1() {
    init(proc_adr(), 2000, 1 0);
}

void (*task1::proc_adr())() {
    return thread_proc;
}

```

The thread procedure must then be defined. This procedure usually comprises a loop which tests whether the object has requested the thread to stop. Stopping the thread must be achieved by calling `stop` - a thread procedure may not return.

```

static void task1::thread_proc() {

    while(!destroyed()) {
        conout << "In Thread 1\n";        Delay(0);
    }
    conout << "Thread 1 Killed\n";        stop();
}

```

# Class directory

The four task and process oriented classes, task, semaphore, event and critical region, are declared in task.hpp. The C library header process.h is automatically include by task.hpp.

## Class task

---

### public member functions

```
void start();
int task_id();
int active();
void kill();
virtual ~task();
```

### protected member functions

```
static void stop();
static int destroyed();
static void set_priority(unsigned);
static unsigned get_priority();
void (*proc_adr())();
void init(void (*func)(),
          unsigned stack_size,
          unsigned _priority,
          int start);
```

## class semaphore

---

### public enumeration types

```
enum sem_state {
    _clear=0,
    _set=1,
    _failed=-1
};
```

### public constructors and destructors

```
semaphore();
virtual ~semaphore() {}
```

### public member functions

```
sem_state set();
sem_state clear();
sem_state wait(int _msecs-1);
sem_state set_and_wait(int _msecs);
sem_state request(int _msecs);
```

## class critical\_region

---

### public constructors and destructors

```
critical_region();  
critical_region(semaphore*);  
virtual ~critical_region();
```

### public member functions

```
void begin();  
void end();
```

### protected member functions

```
semaphore * get_sem();
```

## class event

---

### public constructors and destructors

```
event();  
event(semaphore*);  
virtual ~event();
```

### public member functions

```
void wait();  
void trigger();  
void notify();  
void reset();  
int awaited();
```

### Protected member functions

```
semaphore * get_sem();
```

# *Class reference*

## **class task**

---

### **Constructors**

The task class is an abstract class. The constructor for the user derived class must call protected member function `init`. See Task introduction.

### **Public member functions**

The following member functions may be called using an object of type class, but may not be called by the thread procedure itself.

```
void start();
```

Starts thread if not already started.

```
int task_id();
```

Return thread number of the thread associated with the object

```
int active();
```

Returns non-zero if the thread is running.

```
void kill();
```

Sends signal to thread to stop. This function does not actually cause the thread associated with the object to terminate, but a call to `destroyed()` by the thread procedure will return true following a call to `kill()`.

The following static member functions can only be called by the thread procedure.

```
static void stop();
```

Stops execution of the thread.

```
static int destroyed();
```

Returns non-zero a call to kill associated object has been made.

```
static void set_priority(unsigned);
```

Sets thread priority.

```
static unsigned get_priority();
```

Returns thread's current priority.

## **Protected member functions**

The following protected member functions may be used by the derived class's constructor:

```
void (*proc_adr())();Returns the address of the thread procedure
associated with the object.
void init(void (*func)(),
                                unsigned    _stack_size,
                                unsigned _priority,
                                int _start);
```

Initializes the thread object with the thread procedure address, desired stack size, initial priority and state. If the stack size parameter is not specified 2000 bytes will be allocated. If the priority parameter is not specified an initial priority of 1 will be selected. If no start flag is specified a value of 0 will be used; the thread will not be started during the constructor call.

## **class semaphore**

---

### **public enumeration types**

```
enum sem_state {
    _clear,
    _set,
    _failed
};
```

The return value of most member functions is of this type. `_clear` indicates that the semaphore was previously clear, `_set` indicates that the semaphore was previously set and `_failed` indicates that the operation failed. A `_failed` return value is normally caused by the failure to acquire the semaphore before the specified number of milliseconds expired.

### **constructors and destructors**

```
virtual    ~semaphore();
semaphore();
```

### **public member functions:**

```
sem_state set();
```

Unconditionally sets semaphore. The previous state of the semaphore is returned.

```
sem_state clear();
```

Unconditionally clears semaphore. The previous state of the semaphore is returned.

```
sem_state wait(int _msecs);
```

Waits `_msecs` milliseconds for the semaphore to be cleared. The default value of -1 for `msecs` will cause the thread to wait indefinitely for the semaphore to be cleared.

```
sem_state set_and_wait(int _msecs);
```

Unconditionally sets semaphore and then waits `_msecs` milliseconds for the semaphore to be cleared. The default value of -1 for `msecs` will cause the thread to wait indefinitely for the semaphore to be cleared.

```
sem_state request(int _msecs);
```

Waits `_msecs` milliseconds for the semaphore to be cleared and then sets it. The default value of -1 for `_msecs` will cause the thread to wait indefinitely for the semaphore to be cleared.

## class critical\_region

---

### **Constructors**

```
critical_region();  
critical_region(semaphore*);
```

The constructor may be called with no parameter in which case a semaphore object will be created dynamically, or with the address of an existing semaphore.

### **Public member functions**

```
void begin();
```

Begins critical region. Any other thread attempting to begin the same critical region will wait.

```
void end();
```

Ends critical region. Any thread waiting to enter the region will continue when it is re-scheduled providing another thread has not entered the region.

### **Protected member functions**

```
semaphore * get_sem();
```

The protected member function returns the address of the semaphore associated with the critical region.

## class event

---

### **Constructors**

```
event();  
event(semaphore*);
```

The constructor may be called with no parameter in which case a semaphore object will be created dynamically. If the address of an existing semaphore is provided it that semaphore will be used to control the event..

## **Public member functions**

```
void wait();
```

A thread calling wait() will wait until the event is triggered.

```
void trigger();
```

The event is triggered. If the event count is negative the semaphore associated with the event is cleared. The event count is incremented and any threads waiting for the signal may continue.

```
void notify();
```

The event is triggered only if another thread is waiting. If the event count is negative the semaphore associated with the event is cleared, and the event count is then incremented.

```
void reset();
```

The signal is reset to its initial state. The effect of resetting an event while other threads are waiting is undefined.

```
int awaited();
```

Returns true if another thread is waiting for the signal.

## **Protected member functions**

```
semaphore * get_sem();
```

The protected member function returns the address of the semaphore associated with the event.

## **CHAPTER 5**

# **WINDOW CLASSES**

The `window` and `palettewindow` classes provide C++ bindings to the JPI window module, (see C Library Reference). Any objects of these classes are fully compatible with windows opened using the JPI window module. This documentation is written with the assumption that the user is familiar with C library window and console I/O functions.

Neither class has input or output functions. The C library console I/O functions and the predefined iostreams `conin` and `conout` are vectored to the currently active window selected by `window.use` or `window.putontop`.

The class `palettewindow` is derived from class `window` and allows foreground and background colors to be changed dynamically.

The two window classes are declared in `textwin.hpp`. The JPI window module header `window.h` is automatically included.

The predefined streams `conin` and `conout` are declared in `coniostr.hpp`. These streams may be used for console input and output whether or not the JPI window module is active.

The C console I/O function `cprintf`, `cputs` etc. may also be used.

## ***Class Directory***

Declarations and definitions for classes `window` and `palettewindow` are found in `textwin.hpp`.

Declarations and definitions for class `complex` are found in `complex.hpp`.

The declaration of the predefined streams `conin` and `conout` are found in `coniostr.hpp`.

### **class window**

---

#### **Constructors**

```
window(window& W);
window(wintype WT);
window(windef& WD);
window(abscoord X1,
        abscoord Y1,
        abscoord X2,
        abscoord Y2,
        Color fore,
        Color back);
window(abscoord X1,
        abscoord Y1,
        abscoord X2,
        abscoord Y2,
        framestr frame,
        Color framefore,
        Color frameback,
        Color fore,
        Color back);
```

#### **Public operators**

```
void operator=(window& W);
```

**Public member functions**

```

void  settitle(char _NewTitle[],
              TitleMode _Mode );
void  setframe(framestr _Frame,
              Color _Fore,
              Color _Back);
void  use();
void  putontop();
void  putbeneath(window&);
void  hide();
void  change(abscoord _X1,
              abscoord _Y1,
              abscoord _X2,
              abscoord _Y2 );
void  close();
int   isused();
int   istop();
int   obscuredat(relcoord _X,
                 relcoord _Y);
void  convertcoords(relcoord _X,
                   relcoord _y,
                   abscoord& _X0,
                   abscoord& _Y0);
void  info(windef& _WD);
wintype handle();

```

**protected member functions**

```

void check_win();

```

**protected data members**

```

wintype h;

```

**Error handler procedure variable**

```

extern int (* WinErrHandler)()

```

**Default full screen object**

```

extern window FullScreen;

```

## **class palettewindow: window**

---

### **Constructors**

```
palettewindow(PaletteDef _Pal,  
              windef& WD);  
Palettewindow(PaletteDef _Pal,  
              abscoord X1,  
              abscoord Y1,  
              abscoord X2,  
              abscoord Y2,  
              Color fore,  
              Color back);  
palettewindow(PaletteDef _Pal,  
              abscoord X1,  
              abscoord Y1,  
              abscoord X2,  
              abscoord Y2,  
              framestr frame,  
              Color framefore,  
              Color frameback,  
              Color fore,  
              Color back);
```

### **public member functions**

```
void setpalette(PaletteDef _Pal);  
int  palettecolorused(int _pc);
```

### **Predefined input and output stream classes**

```
extern istream conin;  
extern ostream conout;
```

# Class Reference

## class window

---

### Constructors

```
window(window& W);
```

A new window object is constructed. Its size parameters and other attributes are identical to that of the object W except that when the new window is opened the cursor co-ordinates are set to 1, 1 and the window associated with the object becomes the active window.

```
window(wintype WT);
```

A new window object is constructed but a new window is not created. The object becomes an alias for the window identified by the handle WT.

```
window(windef& WD);
```

A new window object is constructed. Its size parameters and other attributes are specified by the contents of the descriptor structure WD. When the new window is opened the cursor co-ordinates are set to 1, 1 and, if the Hide member of WD is false (zero) the window associated with the object becomes the active window.

```
window(abscoord X1, abscoord Y1,
        abscoord X2, abscoord Y2,
        Color fore, Color back);
```

A new object with an unframed window is constructed. Its top left corner is specified by the parameters X1, Y1; its bottom right corner by X2, Y2. fore and back specify foreground and background colors. When the new window is opened the cursor co-ordinates are set to 1, 1 and the window object becomes the active window.

```
window(abscoord X1, abscoord Y1,
        abscoord X2, abscoord Y2,
        framestr frame,
        Color framefore,
        Color frameback,
        Color fore, Color back);
```

A new object with a framed window is constructed. Its top left corner is specified by the parameters X1, Y1; its bottom right corner by X2, Y2. fore and back specify foreground and background colors. The frame pattern is specified by framestr, and its colors by framefore and frameback (see C Library Reference). When the new window is opened the cursor co-ordinates are set to 1, 1 and the window object becomes the active window.

### Destructor

```
virtual ~window();
```

When the destructor for a window object is called the associated window is closed. If the window object was an alias for a previously opened window that window handle becomes invalid.

### **Assignment operator**

```
void operator=(window& W);
```

The window associated with the object is closed and a new window opened. Its size parameters and other attributes are identical to that of the object W except that, when the new window is opened, the cursor co-ordinates are set to 1, 1 and, if the window associated with W was not hidden, the window object becomes the active window.

### **public member functions**

```
void settitle(char NewTitle[],
                                     TitleMode Mode
);
```

Sets the title of the window to the string specified by NewTitle. Its positioning is controlled by Mode, see C Library reference.

```
void setframe(framestr _Frame,
                                     Color _Fore,
                                     Color _Back);
```

Sets the frame of the window to the string specified by Frame, using Fore and Back for foreground and background colors respectively, see C Library reference.

```
void use();
```

Causes output to be vectored to the window associated with the object.

```
void putontop();
```

Puts the window associated object on top of the window stack and causes output to be vectored to that window.

```
void putbeneath(window& W);
```

Puts the window associated with the object beneath the window associated with W.

```
void hide();
```

Hides the window associated with the object.

```
void change(abscoord _X1, abscoord _Y1,
                                     abscoord _X2, abscoord
_Y2 );
```

Changes the top-left and bottom right co-ordinates of the window associated with the object.

```
void close();
```

Closes the window associated with the object. Any attempt to use the object will cause a run-time error and a call to the window error handler, if

installed. The window may only be reopened by assigning a currently open window object.

```
int isused();
```

Returns non-zero if the window associated with the object is currently used for output.

```
int istop();
```

Returns non-zero if the window associated with the object is at the top of the window stack.

```
int obscuredat(relcoord _X, relcoord _Y);
```

Returns non-zero if the window associated with the object is obscured by another window at the specified relative co-ordinates.

```
void convertcoords(relcoord _X,
```

```
relcoord _y,
```

```
abscoord& _X0,
```

```
abscoord& _Y0);
```

Converts the relative co-ordinates for the window to absolute screen co-ordinates.

```
void info(windef& _WD);
```

Information on the state and parameters of the window object is stored in the window descriptor WD.

```
wintype handle();
```

Returns the window handle associated with the object. This handle may be used as a parameter for the C or Modula-2 JPI window module functions.

### **protected member functions**

```
void check_win();
```

Checks the validity of the window associated with object. If the window is not open or initialization has failed the error handling procedure will be called, if installed.

### **protected data members**

```
wintype h;
```

The handle for the associated window.

### **Error handler procedure variable**

```
extern int (* WinErrHandler)();
```

The procedure pointed to by WinErrHandler will be called if an unopened window is used or an error occurs during a constructor call. If the error handling procedure returns non-zero program execution will continue, otherwise the program will terminate with error code 0X30.

## **Default full screen object**

```
extern window FullScreen;
```

All output from console I/O functions and conout will be directed to the default window object FullScreen unless another object is currently active. The window associated with this object corresponds to the full screen at program startup.

## **class palettewindow: window**

---

### **Constructors**

```
palettewindow(PaletteDef Pal, windef& WD);
```

A new palette window object is constructed. Its size parameters and other attributes are specified by the contents of the descriptor structure WD. The initial palette is specified by Pal. When the new window is opened the cursor co-ordinates are set to 1, 1 and the window object becomes the active window.

```
palettewindow(PaletteDef _Pal,
               abscoord X1,
               abscoord Y1,
               abscoord X2,
               abscoord Y2,
               Color fore,
               Color back);
```

A new object with an unframed window is constructed. Its top left corner is specified by the parameters X1, Y1; its bottom right corner by X2, Y2. fore and back specify foreground and background colors. When the new window is opened the cursor co-ordinates are set to 1, 1 and the window object becomes the active window. The initial palette is specified by Pal.

```
palettewindow(PaletteDef _Pal,
               abscoord X1,
               abscoord Y1,
               abscoord X2,
               abscoord Y2,
               framestr frame,
               Color framefore,
               Color frameback,
               Color fore,
               Color back);
```

A new object with a framed window is constructed. Its top left corner is specified by the parameters X1, Y1; its bottom right corner by X2, Y2. fore and back specify foreground and background colors. The frame pattern is specified by framestr, and its colors by framefore and frameback (see C Library Reference). When the new window is opened the cursor co-ordinates are set to 1, 1 and the window object becomes the active window. The initial palette is specified by Pal.

**public member functions**

```
void setpalette(PaletteDef _Pal);
```

Sets the palette to that specified by \_Pal.

```
int palettecolorused(int _pc);
```

Returns the color used for the palette color number specified by pc.

**Predefined input and output stream classes**

```
extern istream conin;  
extern ostream conout;
```

The predefined streams vector input and output to the currently active window if the JPI window module is active or to the console if no window module is active.

# CHAPTER 6

## STREAM CLASSES

The stream classes in your TopSpeed Library meet three major requirements of an input-output interface:

- They provide device-independent input-output;
- They provide type-independent input-output without dropping compile-time type checking;
- They can be extended to new types without re-implementing any stream members.

The importance of these objectives can be understood best by comparing C++ streams with C language i/o as provided for in the ANSI standard functions.

### Weaknesses of C input-output

---

#### Waste

In the ANSI C library there are three separate functions, with the same syntax, for each i/o operation:

- Functions like `printf` and `scanf` deal with input and output through devices such as the screen and the keyboard.
- Functions like `fprintf` and `fscanf` deal with file i/o.
- Functions like `sprintf` and `sscanf` deal with incore formatting, which looks like i/o but writes to and reads from an area of ordinary memory.

This wastes code and makes the programmer's job harder. It is notoriously difficult to write general-purpose i/o functions in C. A function which writes to a device cannot write direct to memory. Just to alternate device and file i/o, the program must be written to use the operating system's redirection facilities. By relying on the operating system to supply redirection, the program is reducing its portability.

## **Inadequate type checking**

`printf` is the first C function which any programmer encounters. It is a bad example; it bypasses at least two important standards:

- Type-checking is turned off, so that it can be called with an argument of any type;
- It has a variable number of arguments.

Why does C abandon its own standards? The alternative is even greater duplication: a separate function for the input and output of each main data type.

## **Problems extending to new types**

It is virtually impossible to extend the C library i/o to new types such as structures and arrays. This is not just because it is hard to write general-purpose i/o procedures. The library functions themselves are ‘closed off’ and cannot be extended. `printf` will only handle predefined types, and the only way to print a new type is to rewrite `printf` or create a new function.

## **The C++ solution**

---

There is no simple way round these problems in C. ANSI C input-output is limited by C itself.

The stream library tackles these problems by drawing on C++’s power of abstraction. All three of the i/o operations described above have two elements in common:

- They convert between objects and their character representation according to well-defined formatting rules;
- They process an ordered sequence of objects.

The stream classes generalize from these two characteristics. Once you have specified formatting rules, and laid down how the operations should be sequenced, the result can be applied to any object which provides or accepts characters in sequence.

In large measure this overcomes the limitations of C’s input-output:

- Streams provide type-independent input-output without switching off compile-time type checking. The same syntax is used for the input and output of any object, and compile-time type-checking is applied, not just to ensure that i/o statements are meaningful but — through the device of overloading — to decide what they mean.
- Streams offer genuinely device-independent input-output. Any device can be represented by a C++ stream, so that the same i/o

statement can deal with a screen, a printer, a file or a keyboard. If there are device-specific features — for example, if the device is interactive — these can be dealt with where appropriate by creating derived streams which inherit from the standard C++ streams, so that all existing stream i/o will work with the new device.

- The stream classes facilitate user-defined extensions so that they can be redefined, specialized or adapted to suit the application's needs.

The problem is a difficult one and the stream classes have improved over time. The first version of stream classes — release 1.2 — solved most of the problems above, but did not fully integrate incore i/o into its general framework. This has been overcome in the new release.

There are still weaknesses, which reflect design limitations in C++ itself. For example, though a stream type is determined at run-time, the class of the object being output is fixed at compile time. This is efficient, but it means that the power of C++'s virtual function mechanisms cannot be used to the full. A collection of objects of arbitrary type cannot easily be sent to a stream.

Nevertheless the release 2.0 classes represent a major improvement over the input-output facilities offered by ANSI C, and will reward the investment in time of study and use.

## ***The stream class hierarchy***

The stream classes have a quite developed hierarchy. This is listed in full at the beginning of the stream class directory; a brief overview may be useful at this point, but you can skip it and return to it later if you prefer to move directly to examples of stream usage.

### **The ios class and the streambuf**

---

The two base classes of the hierarchy correspond to the common elements we explained in the previous section:

- The ios class provides format control and condition handling;
- The streambuf class provides sequencing.

Any ios instance contains a streambuf member which it gets characters from or sends characters to. Specialist streams are derived from class ios, and usually contain corresponding specializations of streambuf.

#### **istreams and ostream**

The first basic division of the class hierarchy is between output streams and input streams. Class istream forms the base class for all input streams and class ostream the base for all output streams. A special class, class iostream, can do both input and output. It is a marriage of convenience between istream and ostream and inherits from both. Since it inherits from ios via both bloodlines, istream and ostream use ios as a virtual base class to avoid duplication.

#### **Specializing a stream**

Specialized stream classes deal with file i/o and with ‘incore’ i/o. Each is derived in the same way:

- a specialist streambuf is derived to handle the device concerned - for file i/o this is called filebuf, while for incore operations it is strstreambuf;
- a specialized ios is derived to handle any operations specific to the new streambuf — for file i/o this is fstreambase, which handles operations like open, while for incore operations the class strstreambase handles the allocation of buffer space;
- an input and output stream class is derived which inherits from istream or ostream, and from the specialized ios derivative, and which uses the new streambuf class as its source or sink for characters. For file i/o these new classes are ifstream and ofstream respectively; for incore operations istrstream and

ostream.

- an input-output class is derived from ostream and from the new ios derivative.

### **Assignable stream classes**

Unlike the old release 1.2 classes, the new stream types cannot be assigned. A further derivation of ostream is called ostream\_withassign, and an assignment operator is defined for this class so it can be redirected. These assignable classes connect to the standard input and output by default, so they can be used to define standard keyboard and screen streams.

# Elementary input-output

## The operators << and >>

---

The two basic stream classes are ostream and istream: the output and input streams respectively. A stream can be a device, a file or an area of memory.

C++ provides four predefined streams, which can be reassigned by the user: cin, cout, cerr, and clog. The first three are the standard input, standard output, and standard error streams respectively. (clog writes the same error stream as cerr but is buffered).

An object of any basic type can be sent to any ostream with the insertion operator <<. It can be read from any istream using the extraction operator >>.

Thus to write a character string to the output screen, write

```
cout << "Hello World";
```

To read an integer from the keyboard, write:

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

To read a double from the keyboard, write:

```
double f; cin >> f;
```

and so on.

**Note:** Ordinary istream and ostream instances cannot be assigned to. A specialization of istream, istream\_withassign, permits assignment. See Stream Class Reference.

## Operator consistency

Operators are defined similarly to assignment operators so that they can be stacked:

```
cout << "Hello " << "World";
cin >> i >> d;
```

This makes it quite natural and simple to format output by sending a succession of objects to a stream:

```
cout
  << "Sum of "
  << x
  << " and "
  << y
  << " is "
  << x + y;
```

If x is 2 and y is 3 this would produce the character sequence

“Sum of 2 and 3 is 5”

Input can also be chained, but it is more complex unless you can guarantee that the input is correct. In general input procedures should be able to deal with wrongly-formatted input.

Some simple input formatting is provided. The standard input functions skip whitespace (tab, space and return) characters when asked to read a simple type. When applied to the input “A = 1” the statements

```
char c, d;
int i;
cin >> c >> d >> i;
```

would set `c` to ‘A’, `d` to ‘=’ and `i` to 1.

For more complex cases the user has to define her or his own procedures. The stream classes provide many member functions, and very sophisticated condition-handling facilities, for precisely this purpose.

## Character (unformatted) reading and writing

---

The stream class character functions supplement the `<<` and `>>` operators as building blocks for user-defined stream operators and functions. They offer more detailed control than the simple inserters and extractors, but they simply transfer characters rather than interpreting or translating them. For this reason they are generally described as unformatted functions.

Class `ostream` provides two member functions for character writes:

```
ostream& put(char c);
ostream& write(char* address, int num);
```

The first sends a single character to the stream; the second, which is mainly used for binary output, sends a string of characters of maximum length `num`, without the null terminator.

**Note:** the `write` member is more complex because it is overloaded to take both `signed` and `unsigned char*` types, and because the `char*` parameter has a `const` qualifier. In this introduction the sophistication is not important and we shall omit it.

**istream** provides a wider range:

```
istream& get(char&);
int get();
istream& get(char*, int, char = '\n');
istream& getline(char*, int, char = '\n');
istream& get(streambuf&, char = '\n');
istream& read(char* address, int num);
```

These are dealt with in full in the reference section. The more complex functions read until a specified delimiter character is reached or a specified number of characters has been read. `read`, the mirror image of `write`, is

mostly used for binary input and reads a maximum of num characters starting from address.

### **Return value conventions**

Like the << operator, these functions generally return a reference to ostream. Calls to them may therefore be chained and mixed in with << operations thus:

```
cout.write(c) << x;
```

## **Moving upstream and downstream**

---

Stream classes, unlike moving fingers, can sometimes be lured back to cancel things. Provided it is logically and physically possible, a class may position itself anywhere in a file and start reading or writing there.

This is dealt with in more detail in the section on filestreams, and in the section on streambufs under “Building your own streams”. However two simple functions cover many cases. They provide simple nondestructive reads so that programs can look at what they find in the input stream but leave it as they found it if it is unusable:

```
int peek();
istream& putback(char);
```

The peek function is a non-destructive read, so that successive calls to it just yield the same character. The putback function steps back to the last character that was read, but can actually replace it by another one.

## **A simple example**

---

The following example illustrates some of the functions above. It checks to see if the next character is a string quote: if it is, a string of characters will be read up to but not including the enclosing string quote, to a maximum of 10 characters:

```
char buf[11];  char *p;
char d;
int i;

cin >> d;
if (d != '"') {
    cin.putback(d);
    return;  // with error - no string found
}
for (i = 10, p = buf;
     i && (cin.peek() != '"');
     i--, bufp++)
    cin.get(*p);
*bufp='\0';
```

The function will work whether or not the string quote is put back. The advantage of the technique used here, apart from conforming to the general convention for extractors which fail, is that once the program knows that it is not reading a string, it can call a different extractor which may make more sense of what is present in the input stream.

More advanced techniques use the positioning operations provided by the `streambuf` class, which will be dealt with later.

## ***More sophisticated input: exception handling***

As might be expected, the first complications arise in dealing with unreliable input.

Input is always more complex than output. It is where a program meets the outside world, which it does not control.

C++ deals with this brusquely if bureaucratically: anything it cannot handle is treated as a fault in reality. C++ streams bring two sets of possible conditions under a common roof:

- device error conditions: when the device or file itself cannot be read because characters are exhausted, or because of a mechanical error;
- input format errors: when the device is working, but the input does not conform to what the program expects.

After a device error no more data can usually be read. Format errors, however, may not even be errors. They may simply signal that some special action must be taken. For example they can arise when scanning input for a keyword or special character.

### **The stream state**

---

To make this possible the stream classes maintain a stream state variable which can be interrogated by the program and also written to. The state variable is not reset after each successive read, so that errors propagate from one input or output action to the next.

The mechanisms for error handling are supplied by the base class `ios`, which all stream classes inherit from. The state is actually represented by a private flag word inherited from `ios`. There are `ios` member functions which interrogate the state, just as C functions like `feof` can interrogate a C file handle. Thus

```
ios::eof();    //end of file reached
```

is inherited by all stream classes, so

```
cin.eof()
```

will return true if the stream is exhausted. A further function

```
ios::bad();    //device failure encountered
```

returns true if an external condition prevents further reading — for example if a device has failed or an invalid read request has been issued.

Unlike C, however, the program itself can set the stream state. The result is that the user can diagnose badly-formatted data, tell the stream that

something is wrong, and subsequent attempts to read will pick up this information.

```
ios::fail();    //failure of some kind
```

reports if either an external failure has happened or if a special bit, reserved for user-defined failures, has been set. By convention, processing can continue once the program has dealt with a user-defined failure.

The user-defined fail bit is defined only by convention and the usage made by the ios class member functions. Programs can set stream objects to any state. This is less dangerous than it seems. If a program wrongly asserts that a stream is not at the end of the file, the first attempt to read from it will reset the eof condition.

The ios member function

```
int rdstate();
```

yields the stream state, and a rather misleadingly named member function

```
void clear (int new state= 0);
```

sets the stream state. Conventionally this is done by ORing together values taken from a set supplied by ios as an enumerated type:

```
enum io_state {
    goodbit    = 0x00, //synonym for 'no bit set'
    eofbit     = 0x01, //end of file
    failbit    = 0x02, //user-define fail
    badbit     = 0x04, //external failure
    hardfail   = 0x80, //even worse failure
}
```

Thus

```
cin.clear(cin.rdstate() | ios::failbit);
```

Tells cin that it has failed, without resetting any other conditions that may apply, whereas

```
cin.clear(ios::failbit);
```

Tells cin that it has failed but resets all other conditions.

## A short note about encapsulating objects

---

The stream state is accessed by the functions just described. These are a compromise between efficiency and full encapsulation. In principle, the interface with the stream state should not depend on implementation. It should be possible to re-implement the stream class using, say, a separate variable for each state condition, but rewriting only the state access functions. All such implementations should be link-compatible; you should not have to recompile your application to switch implementations in a fully encapsulated system.

Practice, ever a sordid compromise, has opened the state to the public. It is a flag word in which each bit represents a condition. Moreover the precise mapping between bits and states is in the public domain — defined by the `io_state` enumeration.

The functions which read this state are fully encapsulated — you do not need to know which bit represents an end of file in order to detect the condition. The functions which write to it are not. When you set the stream state you typically write something like:

```
cin.clear(cin.rdstate() | ios::failbit);
```

This reads the existing state using `rdstate()` and then ORs in `ios::failbit` to signal that on top of everything else the input format is suspect. A fully encapsulated implementation would supply a function like:

```
cin.setfail(); // not implemented, sadly
```

There are two practical consequences for the programmer. As will be seen in the section on format control, these become even more significant when dealing with the format state variable:

- Some understanding of the precise layout of the various `ios` class state variables is required. The public enumerated types declared in `ios` supply this information.
- It is important when setting these state variables to follow the conventions outlined in these pages. Given the compromise which the state variables represent, these conventions will probably generate the most portable programs.

A solution to this problem is at hand in the shape of manipulators. These are agents of the stream state; they look like ordinary objects, but they collaborate with the stream to change its state. Manipulators are a programmer-friendly front end for the stream class. Easy to use but complex to write, they offer a route to fully-encapsulated error and format control.

## Input control with stream state functions.

---

The most commonly used state function is

```
ios::good(); //true if stream is ok
```

In addition a `void* ()` operator is defined that calls for some comment:

```
ios::operator void*();
```

If the stream has failed for any reason this returns 0; otherwise it returns a reference to the stream which invoked it. The trick is that according to C++ rules this operator defines an implicit type conversion operator, if a `void *` is required and a stream is supplied. This is used in very typical C++ idiom to repeat a stream read until some end condition is met:

```
while (cin >> i) cout << i;
```

This will copy integers from cin to cout until something goes wrong for any reason. This leads quite naturally to the following type of function, which sums a series of square roots:

```
istream & sumroots(double &total, istream& in){
    while (in)
    {
        in >> x;
        if (x < 0)
            in.clear(
                ios::badbit || in.rdstate());
        else
            total += sqrt(x);
    }
    return in;
}
```

The calling program can test in to find out under what circumstances the function stopped: by calling in.bad it can find out if there were any format errors in the data.

## ***Interactive input-output and flushing***

Input and output are rarely independent. In interactive programs, in particular, user and program form a closed system so that each stream's output is the other's input, an experience which is normally absorbing for both but seldom fulfilling for either.

The important issue in such cases is to ensure that no component of the system is called on to perform logically impossible acts, such as responding to output which has not yet appeared.

This is impaired, as might be expected, by the search for efficiency. streams have a protective attitude to their work and tend to hoard characters so that the ultimate consumer can get them in larger chunks, making less demands on the system.

The easiest way to make sure that everything inserted into an ostream has been sent to the ultimate consumer is to insert a special value, flush. For example:

```
cout << "Please enter date:" << flush ;
```

Inserting flush into an ostream forces all previously output characters to be sent to the ultimate consumer of the ostream. flush is an example of a manipulator (see the previous section), a value that may be inserted into an ostream to have some effect. The last section of this chapter explains how manipulators work and how to write them; but you can happily use them without knowing how they work.

Another useful way to cause flushing is the endl manipulator, which inserts a newline and then flushes.

```
cout << "x=" << x << endl;
```

### **Automatic flushing: the tie function**

---

It is good practice to flush ostreams appropriately. The flush and endl manipulators make it relatively easy to do so. However there are circumstances where automatic flushing is appropriate. This is supported by the ostream\* valued state variable tie.

If in.tie is non-null and in needs more characters, the ostream pointed at by tie is flushed. The standard streams cin and cout illustrate why.

Initially cin is tied to cout so that attempts to get more characters from standard input result in flushing standard output. This ensures that when the program wants input from the user, the relevant requests for input are displayed.

This compromise does not impose a large performance penalty on non-interactive programs because it flushes output only when input is required. It also avoids creating programs which behave differently when connected to a pipe instead of a user.

The overheads are relatively small for the more complex extractors, such as the arithmetic ones, but may be large for single character operations. For this reason it is sometimes a good idea to break the tie by setting the state variable to 0. For example to copy from the keyboard to screen until a control Z is typed one might write:

```
char c ;  
cin.tie(0) ;    //Break tie  
while ( cin.get(c) ) cout.put(c) ;
```

## Format Control

As with C, the application program can vary the default format supplied by the stream class. Though marginally harder to understand than C's format control, C++ format control is a great deal easier to use.

Format is controlled by a format state variable, just as error-handling is controlled by an error state variable. Most format control deals with the layout of numeric types. There are two techniques for changing formats:

- Format functions, which as member functions of the `ios` class are separated out from `<<` and `>>` operations.
- Predefined or user-defined Format manipulators, which can be mixed in with `<<` and `>>` operations.

You can carry out most format control with manipulators without having to worry about the format functions or the detailed setting of the format flag word. This introduction concentrates on the extra detail you will need to understand the internal workings of stream format control, and to use the ordinary format functions.

### Default formats

---

Integer values (except `char` and `unsigned char`) are output in decimal, pointers (except `char*` and `unsigned char*`) in hex, floats and doubles with 6 digits of precision. All are output without leading or trailing padding. `char` and `unsigned char` values are just read as single characters. `char*` and `unsigned char*` values are treated as pointers to strings (null terminated sequences of characters).

The default input format for an integer type is a decimal number. Leading whitespace is permitted. An optional sign (+ or -) is permitted, but without whitespace between it and the digits. Any non-digit character can serve as terminator. A floating point number can be immediately preceded by a sign and must conform to C++ lexical rules.

For many purposes these defaults are adequate. When they are not, the program can do more formatting itself, or it can use the format control features of the stream library.

### The format state variable

---

Any instance of class `ios` contains a format state variable, which is implemented as a long int value that is interpreted as flag word — an array of bits. The position of these bits is declared in `ios` as an anonymous enumerated type:

```
enum{
    skipws      =01, // skip whitespace on input
    left        =02, // left justify if padded
    right       =04, // right justify if padded
    internal    =010, // padding location
    dec         =020, // output numerics as decimal
    oct         =040, // output numerics as octal
    hex         =0100, // output numerics as hex
    showbase    =0200, // (see text)
    showpoint   =0400, // (see text)
    uppercase   =01000, // (see text)
    showpos     =02000, // modifiers
    scientific  =04000, // floats with exponents
    fixed       =010000, // floats as fixed points
} ;
```

Class `ios` also supplies a couple of bit masks to select particular fields in this flag word:

- `basefield` is set to `dec|oct|hex` and selects the fields that control the conversion base;
- `adjustfield` is set to `left|right|internal` and selects the fields that control padding and alignment;
- `floatfield` set to `scientific|fixed` and selects the fields that control floating point format;

Finally the variable width, which is explained in more detail later on, controls field width, and precision controls floating point precision.

## Format functions

---

The basic format control functions are `setf` and `unsetf`. These are usually used to set bits in the flag word explicitly. Thus the statement

```
long ios::setf(long setbits);
```

ORs the format flag word with `setbits`. In other words it will set (turn on) all those bits which are specified in `setbits` but leave the rest of the flag word untouched.

```
unsetf(long setbits);
```

has the reverse effect. It unsets (turns off) every bit which is specified in `setbits`.

A possible use of these two functions is illustrated in the fragment below, which prints a number in octal and another in decimal (a shorter version is given later):

```
cin.unsetf(ios::dec);
cin.setf(ios::oct);
cin << 16;           // will print the string "020"
cin.unsetf(ios::oct);
cin.setf(ios::hex);
cin << 16; // prints the string "0x10"
```

The problem here is that in order to turn decimal formatting on, octal formatting must be turned off: if both `ios::oct` and `ios::hex` were set the effect would be undefined. A longer version of `setf` gives the fine control needed to deal with this problem:

```
long ios::setf(long setbits, long field);
```

The second argument is a bitmask. The format flag word is first ANDed with its complement so that every bit in the format flag word that figures in this bitmask will be turned off, while all others will be left untouched. The first argument is then ORed with the result as in the short version of `setf`. At the end, therefore, all the bits specified by `field` have been turned off except those specified by `setbits`.

This lets us cut the last code fragment in half by removing all the `unsetf` calls:

```
cin.setf(ios::oct, ios::basefield);
cin << 16;
cin.setf(ios::hex, ios::basefield);
cin << 16;
```

Note also that the both versions of `setf` return the previous value of the flag word; however the longer version of `setf` returns only those bits selected by `field`.

Whitespace skipping offers another example. `ios::skipws` controls whether leading whitespace is skipped by extractors. Thus the following fragment would read the next character in line, without skipping whitespace, and would then resume skipping:

```
char c ;
cin.setf(      // turn off skipping
        0, ios::skipws);
cin >> c ;
cin.setf(      //turn back on
        ios::skipws, ios::skipws) ;
```

The problem with this fragment is that if whitespace skipping was already turned off, the code would turn it back on again. A sociable version of this would leave whitespace skipping as it found it. The facility to do this is provided by a very typical function for the stream class which returns the existing flag word and optionally sets a new one:

```
long ios::flags();
```

This yields the format flag word.

```
long ios::flags(long aFlagword);
```

Yields the old flag word and resets the current flag word to be `aFlagword`. The whitespace skipping example can now be written:

```
long f = cin.flags() ;
cin.setf(ios::skipws, ios::skipws) ;
...
cin >> c ;
cin.flags(f) ;
```

This is a very typical pattern which is repeated for other state variables. That is, if `stateVar` is some state variable, and `aStream` is a stream, then `aStream.stateVar()` yields the current value of the state variable and `aStream.stateVar(x)` stores the value `x` into the state variable.

## Field Widths

---

By default `<<` inserts only as many characters as are needed to represent the value, but frequently programs want to have fixed size fields. The following function provides for this.

```
cout.width(5) ;
cout << x ;
```

This will output extra space characters preceding the digits to bring the total number of inserted characters to five.

The `setw` manipulator simplifies this even more. The above code can be written thus:

```
cout << setw(5) << x ;
```

### Padding and truncating

Streams are accommodating. They pad but they never truncate. If the value of `x` will not fit in five characters, enough characters will be inserted to express its value. There is therefore no direct way to specify a maximum number of characters. In cases where a program wants to insert exactly a certain number of characters, it must do the work itself. For example:

```
if (strlen(s) > w)
    cout.write(s,w) ;
else{
    cout.width(w);
    cout << s;
}
```

will always insert exactly `w` characters.

### Defaults quickly restored

The width state variable is reset to 0, inducing the default behavior, whenever it is used. Thus

```
cout.width(5) ;
cout << x << " " << y ;
```

will output `x` in at least five characters, but will use only as many characters as necessary in outputting the separating space and `y`. If this were not done, this fragment would have the annoying effect of outputting at least five spaces between `x` and `y` instead of one.

The value of the width state variable is honored by the library functions, but user defined inserters are responsible for interpreting it themselves.

## Input width

Width is generally ignored by input operators, which tend to rely on the contents of the stream to detect the end of a field. There are a couple of important cases when the value of `width()` can affect input:

- string input into `char*` objects, which is controlled by the value of `width()`;
- the behavior of numeric inserters when whitespace skipping is turned off;

## Inputting strings

The `char*` extractor is a useful operator which requires caution. It takes a `char*` second argument and assumes the program has reserved enough space for the input to fit. For example:

```
char p[100] ; cin >> p ;
```

skips whitespace on `cin`, extracts visible characters from `cin` and copies them into `p` until another whitespace character is encountered. Finally it stores a terminating null character.

The problem is that if there are too many visible characters in the `istream`, the array will overflow.

The width setting can be used to control this: streams interpret a non-zero width to be the size of the array. For example:

```
char a[16] ;
cin.width(sizeof(a));
cin >> a ;
if (!isspace(cin.peek()))
    error("string too long") ;
```

protects the program in case there are sixteen or more visible characters. As a further prophylactic a trailing null is stored in the last byte of the array if extraction stops because there are too many visible characters. This means that the number of characters extracted (not counting leading whitespace) will be at most one less than the specified width.

Using the width operator the earlier example could be more carefully written thus:

```
char p [100] ;
cin.width(sizeof(p)) ;
cin >> p ;
```

**Caution: As a safe programming practice, we recommend that a width is always set when reading `char*` input, especially if whitespace skipping is turned off.**

The `setw` manipulator may be used instead of the `width` member function. The above example is equivalent to:

```
char p[100] ;
cin >> setw(sizeof(p)) >> p;
```

## Numeric extractors, whitespace skipping and width

As a precaution against looping, zero width fields are considered a bad format by input streams. So if the next character is whitespace and `ios::skipws` is not set, the arithmetic extractors will set an error bit.

### Justification

The `ios::left` and `ios::right` flags control whether padding (when it occurs) causes the field to be left or right justified. Since these fields are mutually exclusive, a bitmask `adjustfield` is supplied and should be used, just like the `basefield` mask, in the following type of statement:

```
cout.setf(ios::left, ios::adjustfield);
```

The fill state variable (whose initial value is a space) supplies the character to be inserted when padding takes place. It is set using `fill(fillChar)`. The following code will produce the output “13—,===14”

```
cout.fill('-') ;
cout.setf(ios::left,ios::adjustfield) ;
cout << setw(5) << 13 << “,” ;
cout.fill('='); // set state variable
cout.setf(ios::right,ios::adjustfield) ;
cout << setw(5) << 14;
```

Similarly a fill manipulator, `setfill`, exists to set the fill character. The following code will produce the output “...14”

```
cout << setw(5) << setfill('.') << 14;
```

### Conversion base

Integers are normally output and input in decimal notation, but this is controlled by flag bits. If none of `ios::dec`, `ios::hex`, or `ios::oct` are set, output is decimal but input is interpreted according to the C++ lexical conventions for integral constants.

Manipulators `dec`, `hex` and `oct` provide a convenient abbreviation.

For example:

```
cout
  << dec << 64 << “ “
  << hex << 64 << “ “
  << oct << 64 << “\n”;
```

will produce the output “64 40 100”.

## **Base presentation**

If `ios::showbase` is set then output will convert to an external form that can be read according to these conventions. Thus

```
cout.setf(ios::showbase,ios::showbase) ;  
cout  
    << dec << x << " "  
    << hex << x << " "  
    << oct << x << '\n';
```

will produce “64 0x40 0100”.

## **Floating point precision**

The number of significant digits output by the floating point (double) `>>` operator is controlled by the precision state variable. The details of the conversion are further controlled by various other flags. Consult the reference section for more details.

# *Extending the stream classes to new types*

## **User defined stream operators**

---

As explained at the beginning of this chapter, the stream class has been specifically designed so that it can be seamlessly extended to new class types.

Stream operators can be declared for classes and values of class type and used with exactly the same syntax as for the primitive types.

The simplest kinds of examples are provided by a struct that contains a few values. Consider for example:

```
struct Pair {      int x ;      int y ;      Pair(int=0, int=0);
};
```

The following definition creates an output operator for an object of type Pair which will work with any ostream.

```
ostream& operator<<(ostream& o, Pair& aPair) {
    return o
        << '{'
        << aPair.x
        << " "
        << aPair.y
        << '}' ;
}
```

This will output the pair (0,1) as “{0 1}”.

### **Conventions for user-defined operators**

The declaration above should be seen as a model for user-defined operators and establishes the conventions to be adhered to. The first argument is always a stream reference. The second specifies the object and can be either a pointer, reference or pass by value depending on the nature of the object. The choice is made so as to avoid, if possible, having to reference or dereference the object in the actual call.

The operator returns a reference to a stream, and normally returns the stream which is given to it as the first parameter. This has two effects. Firstly it means that successive << operations can be stacked, so that:

```
cout
    << "Hypotenuse of the triangle on "
    << Pair(3,4)
    << " is "
    << sqrt(x*x + y*y);
```

will output:

```
Hypotenuse of the triangle on {3,4} is 5.0
```

since the result of the first two lines is itself `cout`.

More subtly, it ensures that any changes that have been made to the error or format state will be passed on to the next operator in the list.

User-defined operators should be environmentally-friendly. They should recycle the format state by leaving it as they find it; and if they change the error state they should avoid unsetting previously set flags.

A further useful measure is to build up user-defined operators from the least to the most complex, ensuring that the more complex operators use the less complex ones. If you do this, then any change you make to a low-level operator will propagate through all the high-level operators that use it. To take the example above; if the `Pair` output operator is redefined to print with angle brackets instead of curly ones, then the `triangle` output statement would not require reprogramming.

### **User-defined operators are completely general**

User-defined operators are the clearest illustration of the power of the stream abstraction. If the conventions described above are adhered to, user-defined operators will fit seamlessly into the platform which the stream library provides for the base classes like `char`, `int`, `double` and so on. They will work with any stream class, so that they can be used with devices, files or for incore formatting. They will have no side-effects other than the intended and obvious ones.

As a final and slightly more elaborate example, consider the following class, which is assumed to implement a variable size vector:

```
class Vector {
public:
    Vector() ;
    int   size() ;
    void  resize(int) ;
    float& operator[](int) ;
    ...
};
```

We assume that `Vector` has a current size, which may be modified by `resize`, and that access to individual elements of the vector is supplied by the subscript operator. To output `Vector` values to an ostream, we declare:

```
ostream& operator<< (ostream& o,
                    const Vector& v)
{
    o << "[" ; // use '[' as prefix
    for ( int x = 0 ; x < v.size() ; ++x ){
        if ( x != 0 )
            o << ',' ; // use comma as separator
        o << v[x] ;
    }
    return o << "]" ; // use ']' as suffix
}
```

This will output the list as a comma separated list of numbers surrounded by brackets. The code takes care to get the empty list right and to avoid a trailing comma.

## **User defined Input**

Input operators can obviously be defined in a similar way. However, as explained earlier, input becomes more complex if you want to test the input for integrity.

To take a simple example: suppose we want to read members of class `Pair` defined above. The simplest way to do this would be to insist that the user provides correct input, and read the next two numbers in the input stream into the pair `x` and `y` coordinates. The `Pair` input operator would then be defined thus:

```
istream& operator >> (istream& i, Pair& aPair)
{
    return i >> aPair.x >> aPair.y ;
}
```

The convention is that an input operator converts characters from its first argument, stores the result in its second argument, and returns its first argument. Making the second argument a reference is necessary for class objects which are not themselves pointers, because the aim is to store a new value in them. However the reference mechanism is transparent to anyone using the new operator. A call to the operator would be:

```
Pair p; cin >> p;
```

There is no need to attach an `&` operator to `p` here. This is an improvement on the old C `scanf` input function, when the application had to remember to reference arithmetic objects but not strings. The C++ reference operator makes it possible to do all this work in the operator definition rather than its invocation.

## **Testing input integrity**

The stream class system for dealing with error conditions dovetails effectively into this technique for creating new operators. Suppose that an end of file is reached while reading `aPair.x` in the above operation. The convention followed by the predefined stream operators is that if the stream is in a bad state, they do not modify this state but do not extract any more information. Thus the attempt to read `Pair.y` will not actually do anything, but it will preserve the stream state. The result is that if the whole operation fails for any reason, the program will not crash and no extraneous information will be put anywhere, while the stream can be tested at the end to see what went wrong. With complete safety we can therefore write:

```
while (1) {
    in >> aPair;
    if (!(cin)) break;
    out << aPair)
}
```

which will copy pairs from in to out until the input is exhausted, and will not copy over any incomplete pairs.

It is a good idea to try and adhere to the same conventions when you define your own operators. An input operator whose first argument has a non-zero error state should not extract any more characters from the istream and should not clear bits in the error state. This is not particularly difficult to do: the code in the Pair input operator does nothing explicitly to respect these conventions, but because the only way it modifies in is to use input operations which themselves honor the conventions, the conventions will be respected.

### **Dealing with format conditions**

Format errors fit simply into this framework. A user-defined input operator is allowed to set previously unset error bits, and if it fails for some reason peculiar to the operator, then it should set at least one error bit. As explained earlier, the following conventions exist concerning the meaning of the individual error bits. `ios::failbit` indicates that some problem was encountered while getting characters from the ultimate producer, while `ios::badbit` means that the characters read from the stream did not conform to the expectation of the operator. For example, suppose that the components of a Pair must be input between curly brackets, just as they are output:

```
{0 2}
```

The following operator will read a pair in this format. If the stream ends prematurely it will exit without changing either argument; and if the data does not contain a pair with the correct format it will alter the stream state to include state `ios::badbit`:

```
istream& operator >> (
    istream& in,
    Pair& aPair)
{
    long x,y;
    char c;
    in >> c; if c == '{'
    {
        in >> x >> y >> c;
        if (c == '}')
        {
            aPair.x = x; aPair.y = y;
            return in;
        }
    }
    in.clear(ios::badbit | in->rdstate());
}
```

The Pair extractor has been defined so that it can input values that were output by the Pair inserter. Maintaining this symmetry is an important general principle that is worth some effort.

The final example is a Vector input operator, which requires an opening [ followed by a sequence of numbers, followed by a ]. Recall that the Vector

format uses ‘,’ as a separator and does not insert any whitespace between numbers. The new operator must accept such input. It will also accept slightly more general formats. In particular it allows extra whitespace, and the ‘.’ character to be used as a separator. It also deals properly with a variety of special conditions such as errors in the input format.

```
istream& operator >> (
    istream& in,
    Vector& v)
{
    int n = 0 ; // number of elements
    v.resize(n) ;
    in >> delim ;
    if ( delim != '[' ){ // verify opening '['
        in.putback(delim);
        in.clear(ios::badbit|in.rdstate());
        return in ;
    }
    if ( in.flags() & ios::skipws )
        in >> ws ; // skip whitespace if requested
    if (in.peek() == ']' ) // check for closing ']'
        return in ;
    while ( in && delim != ']' ) // loop
    {
        v.resize(++n) ;
        in >> v[n-1] >> delim ;
        if (delim != ',') // verify separating ','
            in.clear(ios::badbit|in.rdstate());
    }
    return in;
}
```

## A footnote on assignable streams

---

In the old release 1.2 stream library one stream could be assigned to another. In the new stream library this general effect — redirection — is normally achieved by reassigning the streambuf member of a class, because this now deals with all aspects of teaching and carrying characters.

General-purpose assignment is no longer logically possible, because specialized streams may possess member functions that no other stream possesses. As we shall see, a file stream possesses an open function to provoke its filebuf into opening a file. It would make no sense to assign, say, an istrstream to such a stream because an istrstream does not know how to open a file.

Nevertheless, since class istream stands above all other input streams in the class hierarchy, and ostream likewise stands above all other output streams, it is logically reasonable to assign any derivative of istream to an istream or any derivative of ostream to an ostream, provided some control is exercised. An istream does not know how to open a file; therefore if we assign an istrstream to it, the problem described above cannot occur.

The difficulty is that if a general right to assign were granted to `istream`, it would be bequeathed to the entire `istream` dynasty, so that an `ifstream` could be assigned to an `istrstream`, which is what we want to avoid.

The solution adopted is to create a special branch of the `istream` dynasty with an exclusive monopoly on assignment, called `istream_withassign`, and a similar branch of the `ostream` line called `ostream_withassign`. The following declares an assignable `ostream`, and then redirects it first to a `cout` and then to an output file:

```
ostream_with_assign out;
out = cout ;
out << "Hello world";
ofstream
    outputFile("OUTPUT.DAT"); // see next section
out = outputFile;
out << "Goodbye world";
```

The assignment operator for an assignable stream also initializes its state variables. This allows a file static variable of this type (`cout` for example) to be used without being fully constructed, provided it is assigned to first.

## ***File streams***

In the introduction we said that the C++ stream library integrates the treatment of ordinary device-based input-output, files, and incore formatting. The language facility which makes this possible is inheritance. File streams illustrate this point.

File streams are specialized classes which inherit from the base stream classes. As a result, they inherit all the public members and operators which apply to the istream and ostream classes, plus some extra members of their own which are used to control the linkage between the stream and the file concerned on disc.

Classes ofstream and ifstream are derived from ostream and istream and inherit the output and input operations respectively.

### **Opening a file**

---

The simplest way to open a file is to use a constructor which creates and opens it at the same time:

```
ifstream source("IN.DAT") ;
if ( !source )
    error("Couldn't open IN for input");
ofstream target("OUT.DAT") ;
if ( !target )
    error("unable to open 'OUT.DAT' for output");
char c ;
while ( target && source.get(c))
    target.put(c);
```

copies IN.DAT to OUT.DAT. If the ifstream or ofstream constructor is unable to open a file in the requested mode it indicates this in the error state of the stream.

A file stream does not have to be immediately associated with a given physical file. The open member associates it with a file after it has been declared:

```
ifstream file ;
... ;
file.open(answer) ;
```

The file stream variable can be reused by closing and issuing a fresh open call. For example:

```
ifstream infile;
int i;
for ( i=1; i < argc ; i++ ) {
    infile.open(argv[i]) ;
    ... ;
    infile.close() ;
}
```

Files may also be referred to by a low-level file handle, for which the `attach` member is used. By default `cin` is given file descriptor 0, so that:

```
ifstream infile ;
if (fileName)
    infile.open(fileName, input);
else
    infile.attach(0) ;
```

opens `infile` to read a file named by `fileName`, unless the name undefined. In that case it will connect `infile` with the standard input (file descriptor 0).

Closing a file stream, either explicitly or implicitly in the destructor, will close the underlying file descriptor if it was opened with a filename, but not if it was supplied with `attach`. This guards against inadvertently closing the keyboard, an imprudent undertaking fraught with peril.

### **File opening modes**

Files can be opened and used in many different ways. The program may open a new version of the file or look for an existing one; it may overwrite what is there or append to it; and so on. The file stream constructors take a second argument — the mode argument — which provides this information. It replaces the mode parameter in the C `fopen` function. An example is

```
ofstream outfile("OUT.DAT",ios::app|ios::nocreate) ;
```

which declares `outfile` and tries to attach it to a file named `OUT.DAT`. The `ios::app` bit specifies that writes will append to the file. Because `ios::nocreate` is specified the file will not be created. The open will fail if the file does not previously exist.

Possible values of mode are given by an enumerated type of class `ios` called `open_mode`.

```
enum ios::open_mode{
    in,
    out,
    app,
    ate,
    nocreate,
    noreplace }
```

These modes are each individual bits and may be ORed together. Their detailed meanings are described in the reference section.

## **Positioning within a stream**

Since the information in a file exists prior to being read, and persists after it is written, it is both logically and physically possible to move up and down a file stream.

The stream classes provide type `streampos` — which is actually a `long` — to record and define positions in an `iostream`. File streams illustrate this facility.

The stream class member function `tellp` returns a `streampos` value that refers to the current position in the file, while the function `seekg(aStreampos)` restores it.

The following example uses an `fstream`, which is an example of an `iostream` — a class derived via multiple inheritance from both `istream` and `ostream`. This is provided because sometimes it is desirable to use the same file for both input and output.

```
fstream tmp("tmp",ios::in|ios::out);
streampos p = tmp.tellp();
//remember this
tmp << x ;
...
tmp.seekg(p);    //go back where we were
tmp >> x ;
```

This writes `x` to a file, and later returns to the same position to restore the value of `x`.

A variant of `seekg` takes a `streamoff` — also a `long` — to specify an offset from the current position. An enumerated type `seek_dir` specifies the direction. For example:

```
tmp.seekg(-10,ios::end) ;
```

positions the file 10 bytes from the end, and

```
tmp.seekg(10,ios::cur) ;
```

moves the file pointer 10 bytes further than where it is now.

## Binary input and output

---

When external files are used purely as a storage medium, the premium is on the most rapid and efficient means of exporting and importing data. In such cases data is often sent to and from a file without being transformed into its character representation. The `write` and `put` members of class `ostream` provide a simple method for doing this. The first outputs a single character, and the other takes a second parameter which specifies how many bytes should be sent. Single character output is equivalent to using the `<<` operator one character at a time:

```
int c='A' ;
out.put(c) ;
out << (char)c ;
```

The last two lines are equivalent. Each sends a single character 'A' to `out`.

A larger object could be output in binary form using `put` in a loop. It is more efficient to use `write`:

```
out.write((char*)&x, sizeof(x) )
```

will write `sizeof(x)` bytes starting at the first byte of the internal representation of `x`.

**Warning: This operation will not always work as might be expected, and if pointers or virtual functions are involved, it definitely will not. If you want to be certain of mapping a structure in memory onto its equivalent in a file, the rule is that the definition of `x` must be such that it will compile in C, and must contain no pointers.**

Clearly if `x` contains pointers, their integrity will be lost if they are copied to a file. In addition if it contains virtual functions, the compiler will put extra data items in `x` to provide access to these functions.

## Binary Input

The `char` extractor skips whitespace. Programs frequently need to read the next character whether or not it is whitespace. This can be done with the `get` member function. For example:

```
char c;
in.get(c) ;
get returns in. A common idiom is:
char c ;
while (in.get(c))
{...}
```

Programs also need to retrieve binary data, either because it has been written with `write` or `put`, or because a program must map an externally-supplied binary database file into memory. This can be done with `read`, which is the mirror image of `write`.

```
in.read((char*)&x,sizeof(x)) ;
```

The same warnings apply as for `write` above.

For large amounts of binary input it may be more efficient to use the lower level part of the `iostream` library (`streambuf` classes).

## Incore formatting

The power of C++ abstraction is most clearly demonstrated by the `istream` and `ostream` classes which create streams that map onto contiguous areas of main memory. They are derived from `istream` and `ostream` respectively.

Suppose the character representation of an integer is contained in a string buffer, and you want to convert it into an integer `i`. In C you would have to use a special function `atoi`. In C++ you can write:

```
istream(buffer) >> i;
```

This constructs a stream which will read from the area in memory starting at `bufferPointer`.

The argument of the `istream` constructor is a `char` pointer. In this example, there is no need for a named `stringstream`. An anonymous constructor is more direct.

The most important feature of this facility is that it is completely general. Any operation defined for a stream will work for a `stringstream`.

Thus to read a representation of an instance of the `Pair` class defined earlier we simply write:

```
istream(buffer) >> aPair;
```

The same applies to any stream operator. The inverse operation which converts an object `x` into its character representation in a predefined string, is as follows:

```
ostream(buffer, sizeof(buffer))
<< x
<< ends;
```

This will store the character representation of `x` in `buffer` with a terminating null character supplied by the `ends` manipulator.

The constructor is supplied with a `char*` and a size; nothing is ever stored outside bounds defined by these two parameters. In the case above, an output error will occur if an attempt is made to insert more than 32 characters.

## Storage allocation for incore formatting

---

If it is inconvenient to pre-allocate enough space for the string the `ostream` constructor can be used without any arguments. In this case a special member `str` retrospectively allocates memory for the stream. Suppose we want to read the entire contents of a file into memory. The following fragment first reads the information and then reserves the space that was used, locking it out from other functions with the new mechanism.

```
ifstream in("IN.DAT");
strstream incore;
char c ;
while ( incore && in.get(c) )
    incore.put(c) ;
char* contents = incore.str();//reserve space
...
delete contents; // Ours now: we must delete it
```

Until `str` is called, space is automatically increased as more characters are inserted. Next, `str` returns a pointer to the currently allocated space. This also freezes `incore` so that no more characters can be inserted. Up to this point it is the responsibility of the `strstream` destructor to free any space that might have been allocated. After the call the space becomes the caller's responsibility, just as if it had been allocated with `new`.

## Defining manipulators

We have already met the simple manipulators `dec`, `oct`, `hex`, `setw`, `endl`, `ends` and `flush`. As can be seen, though they do not add much functionality they make code a great deal more transparent.

A manipulator is a fake object. It is sent to or extracted from a stream as if it were data, but it has a secret mission: to change the stream state variables. The `hex` manipulator, for example, produces no output but ensures that subsequent numbers will be processed in hexadecimal format.

Programmers can define their own manipulators. Four special member functions, two for ostream and two for istream, make manipulators possible. These are:

```
ostream& ostream::
    operator <<
        (ios& (*sManipulator)(ios &anIos))

istream& istream::
    operator >>
        (ios& (*iManipulator)(ios &anIos))
ostream& ostream::
    operator <<
        (ostream& (*oManipulator)(ostream &os))

istream& istream::
    operator >>
        (istream& (*sManipulator)(ios &is))
```

In what follows we shall refer to these special functions as combinators or operatives to distinguish them from ordinary operators.

The trick is that the second parameter of the operative is itself a function: a function taking a stream and returning a stream. The recipient of the `<<` message collaborates with this deception by invoking `Manipulator` with itself as parameter, so that manipulator can change the state variables. Formally, the result of writing:

```
out << aManipulator;
```

is to invoke

```
aManipulator (out);
```

The `<<` combinator simply recombines the components of the expression in a different order.

A manipulator is therefore any function taking a stream as parameter and returning a stream, or alternatively taking and returning an `ios` reference. Manipulators can be defined just at the `ios` level, or specialized to `ostreams` or `istreams`. Later we shall see that parameterized manipulators can be further specialized to `iostreams`.

To take the simplest case consider a manipulator to insert a tab:

```
ostream& tab(ostream& out) {  
    return out << '\t' ;  
}
```

Now consider the effect of the following call:

```
cout << x << tab << y ;
```

Here `cout` recognizes `tab` as the function `tab` defined above and simply calls it, outputting a tab.

Perhaps a inefficient way to output a tab, but not without advantages. Because `tab` is defined as a function, it will not get confused with any variables or enumerated types of the same name. It also makes code more transparent. Compare the two formulations:

```
cout << hex << x;  
cout.setf(ios::hex, ios::basefield);  
cout << x;
```

## Parameterized manipulators

---

It is common for the function that manipulates a stream to need an auxiliary argument. An example is the `setw` manipulator. This takes a parameter which specifies what the width should be set to.

Unfortunately at this point some of C++'s nastier limitations start surfacing. We can illustrate the problem by trying to parameterize a simple manipulator to print a hexadecimal number and restore the stream so it continues printing in decimal. We start from a first stab which is not parameterized:

```
ostream& printHex(ostream &out){  
    out.setf(ios::showbase, ios::showbase);  
    out.setf(ios::hex, ios::basefield);  
    out.width(10);  
    return out;  
}
```

This sets up a stream to print a hexadecimal value. It could be used thus:

```
cout << printHex << aHexValue;
```

However this does not reset the stream after use. If we wanted to reset the stream we would have to define two further manipulators for this purpose and write something like:

```

cout
  << saveState
  << printHex
  << aHexValue
  << restoreState
  << aDecimalValue;

```

The `printHex` manipulator would be more socially acceptable if took over the job which `saveState` and `restoreState` do in the above sequence. An ideal `printHex` should save the current state, set the stream up for hex printing, output a value and then restore the previous stream state. In use, it should look like this:

```
cout << printHex(aValue);
```

The first step is to create an ordinary overloaded function which does the same job. This is relatively easy to define:

```

ostream& printHex(ostream& out, int n ) {
    long f = flags(ios::showbase|ios::oct) ;
    out << setw(6) << n ;
    flags(f) ;// restore original flags
    return out;
}

```

As an ordinary function this works. The following prints the number 16 as the string “0x0010”

```
cout.printHex(cout, 16);
```

This gives us all the functionality we need, but it cannot be stacked with other `<<` operations. To do this it must be turned into a manipulator.

## From function to manipulator

---

Turning a function into a manipulator presents much greater difficulties. Suppose we try to use `printHex` as it is now defined as a manipulator thus:

```
cout << printHex(16); //syntax error
```

This would not compile. The special `<<` operatives we introduced at the beginning of this section cannot collaborate with `printHex` because `printHex` now has the wrong type; it is a function of two parameters returning an `ios`, not a function of one parameter.

The problem is that C++ does not have a true message-passing facility. Its overloading mechanism is a strictly compile-time mechanism. You cannot give a C++ function an arbitrary function as parameter and expect it to work out at run-time which of a number of overloaded versions of the function it ought to invoke.

Perhaps the stream classes could be adapted by defining a new operative to accept the new `printHex` thus:

```
istream& istream::
operator >>
(istream&(*aBigManipulator)(ios& ,int))
```

The problem is to make this procedure a general one. The operative above can only deal with manipulators that take an `int` parameter. In principle there would have to be as many such operatives as there are types. But C++ allows us to create as many new types as we like. To encompass all possible manipulators, very many operatives would be required.

### **Generic manipulators and the `iomanip.hpp` file**

This is overcome with generic macros that can create such operatives to order. The macros are contained in the file `iomanip.hpp`, which you must include if you intend to use parameterized manipulators. `iomanip.hpp` also contains the predefined parameterized manipulators such as `setw`.

Macros bring with them the usual attendant dangers, but in this context they are reasonably safe because their usage is quite restricted. Note also that as an include file `iomanip.hpp` is available in source form. In principle this is open to abuse, but in practice there is little danger since the code is utterly impenetrable.

### **Predefined two-parameter manipulators**

The `iomanip.hpp` file encapsulates two-parameter functions in classes whose names are given by macros. Macros are available for `ostreams`, `istreams`, `iostreams` and `iostreams`. For the `long` and `int` types these classes are predefined. The relevant macro for our purposes is `OMANIP(int)` — which in fact translates to a class called `omanip_int`.

The `OMANIP(int)` class has two members. One is the two-parameter function we want to insinuate into the stream; the other is the value of its second parameter. The class has a constructor to create it from the function and its value.

An (anonymous) call to the constructor looks like this:

```
OMANIP(int)(printHex, 16)
```

This code fragment will create an instance of class `OMANIP(int)`. This will be a structure with two members: one is a pointer to `printHex`, and the other contains 16. The next step is to persuade this object to apply `printHex` — which, remember, is a function with two parameters — to a stream and to 16. How can we introduce this newborn object to the stream? `iomanip.hpp` defines an operative for the purpose. It takes a reference to an instance of class `OMANIP(int)`, and applies the first member of this instance to the stream and the second member.

The following code fragment therefore invokes `printHex` with `cout` and 16 as parameters:

```
cout << OMANIP(int)(printHex, 16);
```

This is still not quite in the required form; the final flourish can be provided by creating an overloaded version of `printHex` that takes an `int` as parameter and produces the right-hand side of the above:

```
OMANIP(int) printHex(int n){
    return OMANIP(int)(printHex, n);
}
```

The user must define this. The original `printHex` manipulator can then be invoked by writing:

```
cout << printHex(16);
```

## Generic macros for types other than `int` and `long`

---

For parameter types other than `int` and `long` the `MANIP(type)` classes cannot be predefined for reasons given above. If parameterized types were implemented — which may happen in future releases — this would not be a problem. What is required is a parameterized class that is equivalent to `omanip_int` or `omanip_long`, but varies according to the type of the base parameter. Then we could create (for example) `OMANIP(complex)` objects whose members would be:

- 1 A function taking an `ostream&` and a `complex` and returning an `ostream&`;
- 2 A `complex`.

The constructor for this would be:

```
OMANIP(complex)(f, zz);
```

where `f` was a function taking an `ostream&` and a `complex` as parameter.

C++ provides a poor person's parameterized type in the shape of a generic macro. Writing

```
IOMANIPdeclare(complex)
```

declares a set of classes that make the above constructor legal, by defining the `OMANIP(complex)` class for us — along with similar functions such as `IMANIP(complex)` for `istream`s, `SMANIP(complex)` for the `ios` class and `IOMANIP(complex)` for `iostream`s.

Suppose, therefore, we wanted a manipulator to read a `Pair` supplied with square brackets instead of the curly brackets which we chose to use for the standard operator `>>`(`Pair&`). We might define the following function to do this:

```
istream& readSquare(ostream& , Pair&);
```

We are at liberty use this as it stands by writing:

```
in.readSquare(aPair);
```

But if we want to use it as a manipulator we must now write:

```
IMANIPdeclare(Pair); //defines the new class
IMANIP(Pair) readSquare(Pair & aPair)
{
    //returns instance of the class
    return IMANIP(Pair)(readSquare, aPair);
}
```

Finally we could call it by writing:

```
Pair thisPair;
cin >> readSquare(thisPair);
```

Note that the use of `IOMANIPdeclare` imposes no initial code penalty because it only defines classes, creating no class instances. The main limitation imposed by using the preprocessor in this way is that the parameter of `IOMANIPdeclare` must be a single identifier. To create manipulators taking complex objects as parameters or taking more than two parameters, therefore, these objects must be encapsulated in a class or a typedef.

# ***Building your own streams***

## **Introduction**

---

None of the classes in the stream library make major changes or additions to the services provided by the `ios` class; in short they share a common mapping between objects and their character representation.

The major differences between derived classes all relate to the place they get their data from, the place they send it to, and/or the order in which they process it. These tasks are all the domain of the `streambuf` member which every stream class inherits from class `ios`. Each special class type has a special `streambuf` for its purpose. The key to defining a new stream class is therefore to define the appropriate `streambuf`.

## **The streambuf class**

---

The `streambuf` class is the second half of the stream class abstraction. Class `ios` deals with formatting; class `streambuf` deals with sequencing. It deals only with characters. It controls the order they are processed and orchestrates the processes which supply and use them.

Typically, therefore, the `streambuf` class supplies services which:

- define and manipulate position in a stream — which move backwards and forwards within it.
- transfer characters from the stream to the process that consumes them and into the stream from the process that produces them.
- raise efficiency by buffering — organizing a part of memory as temporary storage for characters which have left the producer, but have not yet reached the consumer.

The `streambuf` class is very general, so that many specialized classes can be derived from it — for example stacks, queues of various kinds, and so on. The `filebuf` and `strstreambuf` library classes, used for file and incore operations respectively, are both specializations of `streambuf`. A further derived class, `stdiobuf`, co-ordinates with the C standard streams and figures in the definition of the standard streams such as `cout`.

Because of design weaknesses in previous releases of the stream classes, users who derived new `streambuf` classes were obliged to manipulate its internal representation to achieve the functionality they needed. This reduced portability. The new `streambuf` class is fully encapsulated but aims to supply enough services for you to derive further `streambuf` classes without any loss of generality.

## The public and protected interface

---

It follows that the most important feature of class `streambuf` is the interface between it and its derived classes. These interact with it only via a protected interface. The core of this is:

- protected functions supplied by the base `streambuf` class which supplement the public functions, and are available for derived classes to manipulate the buffer more directly, though with relative safety;
- among these protected functions, a number of virtual functions which are used by the `streambuf` class to synchronize producers and consumers of characters. The most important such functions are underflow for `read(get)` operations and overflow for `write(put)` operations. Derived classes supply their own definitions of these virtual functions, which are then invoked by the `streambuf` class instead of its own, default versions.

The protected interface is not the only service provided by the `streambuf` class. The `streambuf` class supplies many elementary functions for manipulating pointers and retrieving characters. These are mostly not protected, so programs can use them without having to derive classes to do so. Indeed, they are often very useful for programming instances of specialized `streambuf` classes such as `filebuf`.

These are described in the public interface. Examples would be low-level file transfer operations, which you can program directly using the `filebuf` class for greater efficiency, or new stream types where the conversion process can be defined using an existing `streambuf` class's low-level character-handling to supply and dispose of characters.

The `streambuf` class is therefore documented, both here and in the reference section, in two parts. We first deal with the public interface, which you need to create and use `streambuf` objects. The second part covers the protected members of the `streambuf` class, which you need to derive new `streambuf` classes.

**Caution:** We advise users writing their own `streambuf` classes to begin by understanding existing, working derived classes, and therefore strongly recommend purchasing the library source supplied with the TopSpeed SourceKit.

## The streambuf public interface

---

A `streambuf`, conceptually, is a sequence of characters and either one or two pointers into that sequence: a `get` and/or a `put` pointer. Streams read via the

get pointer and write via the put pointer. Some streams, principally istreams, can only get. Others, principally ostream, can only put. Others, such as file streams, do both.

The pointers should be treated as representing locations either before or after characters in the sequence, rather than at specific characters. This makes it easier to understand where characters come from and go to, and at what point an exception condition such as an empty buffer will occur.

### **Elementary get and put operations**

The core of a streambuf is a group of elementary functions that consume or produce characters. Nearly all the more complex operations are composed from these atomic building blocks.

These functions are inline; they are efficient. As long as there are characters to process and a place to put them, they gobble them up. Their reaction to a minor setback, however, verges on paranoia: everything they cannot understand is reported as an end of file. They return an int parameter whose value is EOF in the event of any problem. They have just one recourse before they signal doom: they call up underflow or overflow.

These two functions are the source and sink of the streambuf. The first is called when there is nothing to get, the second when there is nowhere to put. When you derive a new class you decide what powers to endow them with, and this defines your interface with the outside world. Thus the standard input underflow goes to DOS for keyboard input; the filebuf overflow pumps a buffer out to disk — and so on.

### **Device independence: why the streambuf abstraction works.**

The underflow/overflow mechanism is dealt with at greater length in the section on the protected interface. The important thing is the end result. If a streambuf is properly implemented, the interaction with the outside world is transparent. The streambuf user doesn't need to know what underflow and overflow do; indeed as protected members they are not even accessible. Public access to any derived streambuf class goes through the elementary put and get operations, whose behavior is defined independent of what lies behind them. This is what makes the streambuf abstraction possible; this is why the same stream can start off using the keyboard and end up using a file.

### **Elementary output**

Elementary output functions are supplied to:

- store characters in the put area;
- move the put pointer.

A put operation replaces the character after the put pointer or — as is usually the case — appends the character if the pointer is at the end. The put pointer is moved past the stored character. There are two output functions: `sputc` stores a single character; `sputn(ptr,n)` stores the `n` characters addressed by `ptr` after the put pointer and moves the put pointer past them. `sputn` returns the number of characters successfully stored. This will only be less than `n` when something has gone wrong.

A loop to send a string to a streambuf, therefore, would look like this:

```
int printOn(
    char *aString,
    streambuf& aStreambuf)
{
    while(*aString)
        if(aStreambuf.sputc(*aString++) == EOF)
            return 0; //failure
    return 1; //success
}
```

## **Elementary input**

Elementary get operations can:

- remove characters from the get area;
- move the get pointer;
- if possible, put unused characters back into the input stream.

Input is as usual more complex than output, and there are more input functions. The most widely used, `sbumpc`, moves the get pointer forward one character and returns the character it just passed over. `sgetc` returns the character after the get pointer but does not move the pointer. `snextc` moves the get pointer and returns the character following the new position.

The fixed-length buffer read function `sgetn(ptr,n)` is the mirror image of `sputn`. There is one pure move function: `stossc` moves the get pointer forward one character but has no return value. If the pointer started at the end of the sequence this function has no effect.

Recall that the special function `sputbackc(c)` moves the get pointer back one character and puts `c` after it. This may not always have a defined effect if the operation is not physically possible: see the reference for detail.

The following is the actual implementation of the `istream::eatwhite` function, which positions the stream at the first non-whitespace character it can find. `_bp` is the ios class internal name for its streambuf member. `isspace` is a C library function which returns true for whitespace characters:

```

void istream::eatwhite() {
    int c;        // must be int to check for EOF
    while(isspace(c=_bp->sgetc())) {
        _bp->stoss(c);
    }
    if(c == EOF)
        setstate(failbit|eofbit);
}

```

This is terse and efficient and illustrates some of the power of the get functions. The calls to sgetc do not advance the put pointer. As a result when a non-whitespace character is reached, the put pointer is not advanced past it. Notice also how the istream uses the value returned in c to check for end of file and set its state accordingly.

A more complex example is the istream >> operator which copies the input stream to a second streambuf supplied as parameter. The ipfx function, short for ‘input prefix’, is explained in the reference section. It is used to test stream integrity before starting.

```

istream& istream::operator>>(streambuf* sb)
{
    int c;
    if(state == goodbit) {
        while(1)
        {
            if(!ipfx0()) //prefix function
                break;
            c = _bp->sbumpc(); //get and advance
            if(c == EOF)
            {
                setstate(eofbit); //input device error
                break;
            }
            if(sb->sputc(c) == EOF) //put and advance
            {
                putback(c); //output device error
                break;
            }
            ++_gcount;
        }
    }
    return *this;
}

```

Here sbumpc, which does advance the get pointer, can be used to retrieve a value for c. But special steps must be taken if a device error occurs on the output streambuf sb, because the offending character could not be written to it. It is therefore put back into the input stream so that some other part of the program can deal with it.

## **Positioning functions**

The streambuf class has an abstract concept of position which does not depend on the underlying device or process. This does not always make logical sense: the beginning of a keyboard stream is not the easiest thing to

find. These functions are therefore device-dependent to a degree, but they usually do what you would expect.

The class implements two types of move: relative and absolute. It supplies two typedefs for the purpose: `streamoff` for relative moves and `streampos` for absolutes. Conceptually a `streampos` returns the value of a get or put pointer, while a `streamoff` records the difference between the value of the pointer at one place in the stream and its value at another. Arithmetic can sometimes be performed with a `streampos`, but it is best to regard a `streampos` simply as a record of a definite file position which can be returned to later.

**Arithmetic is only valid with a `streamoff` if the stream is binary.**

Class `ios` supplies an enumerated type `seek_dir` — short for `seek_direction` — with three values `ios::beg`, `ios::end` and `ios::cur`. These are used in a relative (`streamoff`) calculation to say whether the offset should be relative to the beginning, end or current pointer position respectively.

The position operations can be used for the get pointer, the put pointer, or both: a mode parameter determines which. Mode values are also supplied by an `ios` enumeration. `ios::out` bit set specifies the put pointer and `ios::in` bit set the get pointer. Both can be moved simultaneously.

The following functions reposition the pointer(s) specified by mode:

**Relative:**        `seekoff(`  
                       `aStreamoff,`  
                       `aSeek_direction,`  
                       `aMode)`

**Absolute:**        `seekpos(`  
                       `aStreampos,`  
                       `aMode)`

`streampos(0)` by convention sets everything to the beginning of the file, and `streampos(EOF)` is used as a constant representing an error (impossible stream position).

Not all classes derived from `streambuf` support repositioning. `seekoff` will return `EOF` if the class does not support repositioning. If the class does support repositioning, `seekoff` will return the new position or `EOF` on error.

#### **Buffer flushing**

`sync` is a 'flush' function which usually ensures there are no waiting characters between the producer and the consumer.

#### **Buffer assignment**

`setbuf(ptr, len, i)` offers a buffer for use by the `streambuf` and is dealt with fully under the protected interface section.

**Note:** `setbuf` does not really belong in the public interface: buffer-level manipulation is strictly speaking the province of derived classes. It is a public function for compatibility with the release 1.2 stream classes which did not make the same public /protected distinction.

## The protected interface: deriving new streambuf classes

---

Under the old release 1.2 stream classes users often found they had to ‘break the abstraction’ of the class and make use of the internal implementation for some of the functions they needed for derived classes. One of the major improvements of the release 2.0 stream classes is that this is no longer necessary. The streambuf class offers a procedure for deriving new classes which preserves the encapsulation. The essential steps are:

- Set up a class definition for the new class so that it inherits public from streambuf, giving it access to streambuf’s protected functions;
- Redefine the virtual functions in streambuf, particularly underflow and overflow;
- Make sure that in your implementation you only access the streambuf buffer via its public and protected functions.

You need some understanding of the internal workings of a streambuf, though you will not be relying on them. The core of the streambuf class consists of:

- Up to three private buffer areas into which characters can be inserted or from which they can be fetched.
- One or two private pointers that define where characters are to be put or got, and public functions —already discussed— for getting and putting characters.
- Virtual procedures which fetch and send when get and put have run out — underflow and overflow. These are the interface with the derived class’s implementation-dependent features.
- Virtual procedures for positioning within the buffer which also form part of the derived class’s definition.
- A flushing procedure called sync which will be called when a streambuf is flushed and which, broadly speaking, has to clear the internal buffer so that the consumer and producer are dealing with the same character with no gaps in between.

### Producing and consuming data

The most important, and in many cases the only virtual functions you have to redefine when creating a new streambuf class are the underflow and

overflow functions. underflow is invoked by gets and overflow by puts. Their job is to produce and consume characters.

Producing and consuming should not be confused with putting and getting. A get operation has a producer and a consumer; so does a put. Consider the keyboard, for example, which would normally figure in a get operation: it produces characters and the program consumes them. Equally the screen consumes what the program produces.

A complex stream with both put and get operations may therefore have more than one producer and more than one consumer. Consider a file: if it is being written to, the producer is the program and the consumer is the file — or, as far as the stream is concerned, the operating system. But if the file is being read, the consumer is the program and the producer is the file. It is perhaps more accurate to think of a consumer and producer being associated with an operation rather than the stream as such.

### **The input - output process in a derived streambuf**

Input or output begins for a streambuf when an external program — usually via the ios class — issues a get or put request. The << operator normally issues a put request, and the >> operator normally issues a get request.

If a streambuf is asked to get, then whoever asked for the character will also consume it. The streambuf's job is to find a producer. It first consults the get buffer to see if anything is available. If it is empty, it calls underflow to fill it. The trick is that as a virtual function, it is redefined by the new class. The standard get functions will invoke the redefined underflow, which does whatever is needed to provoke the outside world into supplying some characters. Thus if the keyboard is involved it will ask the operating system for some user input; if a file, it will perform a file read, and so on.

The put operation is almost the inverse of get. When a streambuf receives a put request, it looks at the put buffer, but this time to see if it is full. If so, it calls the overflow function, which is cast in the role of consumer for the put buffer.

### **What overflow and underflow must do**

If there was no buffering, underflow would be called for every get and overflow for every put. Underflow's job would just be to get one character and return it, and overflow to put one character. The only additional requirement would be to report any device errors encountered on the way.

It is possible to make a streambuf 'unbuffered', which means it acts in just this way, and the underflow and overflow operations must be able to handle this setting.

However the complications all arise when there is a buffer. Overflow, for example, must find out how many characters are waiting for dispatch and, if possible, get rid of all of them. It must then inform the streambuf what it has done by resetting its buffers.

This requires a two-way communication about the location, size and state of the buffers. The bulk of the protected interface is concerned with protected functions to supply this communication.

In the release 1.2 streams these functions were not provided, so that underflow and overflow had to deal with the streambuf pointers directly. In the new implementation the pointers are private and can only be manipulated via the protected interface, but the buffer itself is still accessed by derived classes.

### **Streambuf buffer areas**

The streambuf class allocates an area of storage — the buffer or reserve area — as a staging post for characters in transit between producer and consumer. In theory buffering is more complicated since it also allocates a get buffer area and a put buffer area. In practice (and by default) the get and put areas are simply part of the general reserve area, but provision is made for derived classes to allocate any area of memory for each of the three buffers.

A streambuf may have zero length buffers, in which case characters must be consumed as soon as they are produced.

The streambuf user is allowed to supply suggested buffer areas using the `setbuf` and other functions; for details see the reference section.

### **The put and get pointers**

Much of the variety in derived streambuf classes flows from the different ways they interpret the operations of putting and getting.

The simplest buffers, which cover most simple input and output, are objects like the keyboard and screen which permit only gets or permit only puts.

A streambuf of this kind only uses one buffer and one pointer. The keyboard, for example, can only produce; it cannot consume. Therefore it only requires the get buffer and pointer area.

More sophisticated streambuf types include:

- Queue-like buffers such as `strstream` which behave a bit like type-ahead buffers. These have a put and a get pointer which move independently of each other. Characters that have been produced are held until consumed, though a logical or physical limit is placed on the number of characters which can be saved

like this.

- File-like buffers such as `filebuf`. These permit both gets and puts but have only a single pointer. Everything which is written can also be read, and vice versa.

Some of the effects of getting and putting are common to all derivations and are therefore defined in the public interface, but most of the details are left to specialized classes derived from `streambuf`.

### **Pointer representation**

The `streambuf` class actually represents the get and put pointers with a real pointer to the buffer. The user can get access to this directly from within a derived class because it is sometimes necessary to test for things like the number of characters in the buffer, or to inspect the relative position of pointers which overlap, as in a circular buffer where one process puts, and the other gets, to the same buffer.

Nevertheless it is good programming style to try and avoid direct pointer manipulation and is likely to make your application more portable

## **Using streambufs in streams**

---

The positions of the put pointer after operations that store characters, and position of the get pointer after operations that fetch characters are well defined by the sequence abstraction. But the location of the get pointer after stores, and the location of the put pointer after fetches is not. Most specializations of `streambuf` (i.e., classes derived from it) follow one of two patterns. Either the class is queue-like, which means that the put pointer and the get pointer are independent and moving one has no effect on the other. Or the class is file-like, which means that when one pointer moves the other is adjusted to point to the same place. So a file-like class behaves as if there were only one pointer. Other possibilities are logically possible, but do not seem to be as useful.

### **Setbuf**

---

The virtual function `setbuf` is called by user code to offer an array for use as a holding area. It can also be used to turn off buffering.

### **Overflow**

---

The virtual function `overflow` is called to send some characters to the consumer, and establish the put area. Usually (but not always) when it is called, the put area has no space remaining.

## **underflow**

---

The virtual function `underflow` is called when no more characters are available to be consumed, i.e. when the `get` area is empty.

## **sync**

---

The virtual function `sync` is called to maintain synchronization between the various areas and the producer or consumer. It is also called by the `streambuf` destructor.

The virtual functions defined above implement a correct `streambuf` class. A possible refinement would be to provide implementations of the virtual `sputn` and `sgetcn` functions. These functions are called when chunks of characters are being inserted and extracted respectively. Their default actions are to copy the data into the buffer. If they were defined in the `factbuf` class they could call the functions directly and avoid the extra copy.

## **Extending streams**

---

There are two kinds of reasons to extend the basic stream classes. The first is to specialize to a particular kind of `streambuf` and the second is to add some new state variables.

## **Specializing `istream` or `ostream`**

---

When the `iostream` library is specialized for a new source or sink of characters the natural pattern is this: First derive a class from `streambuf`, such as `factbuf` in the previous section. Then derive classes from whichever of `istream`, `ostream`, or `iostream` is appropriate.

Derivations from `ios` are virtual so that when the class hierarchy joins there will be only one copy of the error state information. Because the derivation from `ios` is virtual an argument cannot be supplied to its constructor. The `streambuf` is supplied via `ios::init`, which is a protected member of `ios` intended precisely for this purpose.

## **Extending state variables**

---

In many circumstances we would like to add state variables to streams. For example, suppose we are printing trees and would like to have an indentation level associated with an `ostream`.

```

int xdent = ios::xalloc() ;
    // generate a unique index
ostream& indent(ostream& o) {
    // manipulator that inserts newlines and
    // appropriate number of tabs
    o << '\n' ;
    int count = o.iword(xdent) ;
    while ( count > 0 ) o << '/t' ;
    return o ;
}
ostream& redent(ostream& o, int n) {
    // parameterized manipulator that
    // modifies indentation level
    o.iword(xdent) += n ;
    return o ;
}

```

`o.iword(xdent)` is a reference to the `xdent`'th integer state variable. Each call to `ios::xalloc` returns a different index. The index may then be used to access a word associated with the stream. The reason for calling `ios::xalloc` to get an index rather than just picking an arbitrary one is that it allows combining code that used the indentation level with code that may have extended the formatting state variables for some other purpose.

A subtle problem occurs in the above example because `xdent` is initialized by a function call. What if `indent` or `redent` were called before `xdent` was initialized? Can that happen? Yes it can. It can happen if `indent` or `redent` is called from inside a constructor that is itself called to initialize some variable with program extent. Problems with order of initialization when doing I/O in constructors are common. The solution relies on “tricks” to force initialization order. In this case we would put into the header file containing the declarations of `indent` and `redent`:

```

static class Indent_init {
    static int count ;
public:
    Indent_init() ;
    ~Indent_init() ;
} indent_init ;

```

Each file that includes this header file will have a local variable `indent_init` that has to be initialized. Because this variable is declared in the header its initialization will occur early.

The definition of the constructor and destructor looks like:

```

static Iostream_init* io ;
Indent_init::Indent_init()
{
    // count keeps track of the          difference between how
    // many constructor and destructor    calls there are
    if ( count++ > 0 ) return ;
    // This code is executed only the     first time
    io = new Iostream_init ;
    xdent = ios::xalloc() ;
}
Indent_init::~Indent_init()
{
    if ( -count ) > 0 ) return ;
    // This code will be executed the     last time to delete io ;
}

```

The iostream library uses this idea itself. The constructor for `Iostream_init` causes the iostream library to be initialized the first time it is called. It also keeps track of how many times the constructor is called and will do finalization operations on various data structures the last time it is called. It is therefore important that any values of type `Iostream_init` that are constructed by a program are eventually deleted. This is the purpose of having an `Indent_init` destructor; even though there are no finalization operations associated with indentation, it must delete `io`.

## Creating streams

---

The examples so far have used the predefined streams, `cin` and `cout`. For some programs, reading from standard input and writing to standard output suffices. But other programs need to create streams with alternate sources and sinks for characters. This section discusses the various kinds of streams that are available in the iostream library.

## ***Conversion from 1.2***

### **Converting from streams to iostreams**

---

The iostream library is mostly upward compatible with the older stream library, but there are a few places where differences may affect programs. This section discusses those differences.

The major conceptual difference is that in the iostream library, streams and streambufs are regarded solely as abstract classes. The old stream classes provided certain specialized behaviors, specifically incore formatting and file I/O. In the iostream library these are supported solely through derived classes.

The old stream library declared everything in the header file `stream.hpp`. The iostream library uses `iostream.hpp` and some other headers. For compatibility a `stream.hpp` is supplied that includes `iostream.hpp` and other headers that are required for compatibility and defines a variety of items whose names are different in the iostream and stream libraries.

### **streambuf internals**

---

The internals of the streambuf class in the stream library were all public. Any program that relies on these internals will break because they are different (and private) in the iostream library.

How to derive new streambuf classes was not documented in the stream library. But it is such a natural idea to do so that many programs do it. Converting these programs to the iostream library may require changes in the derived overflow and underflow functions. The functionality of these functions in the iostream library is essentially the same as in the stream library. But because the internals of streambuf have changed, some code changes will probably be required. In particular the code will have to use the (protected) streambuf member functions `setb`, `setg`, and `setp` instead of directly manipulating the pointers.

### **Incore formatting**

---

In the stream library the use of arrays of characters as sources or sinks was supported as the default behavior of streambuf. Although some attempt to preserve the default behavior is made by the iostream library these uses of a streambuf are considered obsolete. The support of incore operations is specifically the responsibility of the `strstreambuf` declared in `strstream.hpp`.

streambufs created for this purpose can usually be replaced directly by strstreambufs that have equivalent behavior. The stream usage:

```
char* buf[10] ;  
streambuf b(buf,10) ;
```

is equivalent to the istream:

```
char* buf [10] ;  
strstreambuf b(buf,10) ;
```

and the old method for initializing a streambuf for extraction:

```
char* buf[10] ;  
streambuf b ;  
b.setbuf(buf,10,buf+5) ;
```

is equivalent to the istream method:

```
char* buf[10] ;  
strstreambuf b(buf,10,buf+5) ;
```

Frequently these uses of streambuf do not appear explicitly in the program but are the consequence of using certain constructors of istream and ostream. These constructors are supplied in the istream library, but are considered obsolete. The equivalent forms using strstream should be used.

The old member of storing a formatted value into an array:

```
char* buf [10] ;  
ostream out(10,b) ;
```

is replaced by:

```
char* buf[10] ;  
ostrstream out(b,10) ;
```

Note that the order of the arguments is reversed. The new order creates more consistency between various uses of strstreams.

The old method of extracting a formatted value from an array:

```
char* buf[10] ;  
istream in(10,b) ;
```

is replaced by:

```
char* buf[10] ;  
istrstream in(b,10) ;
```

The old istream constructor allowed an optional extra argument to specify skipping of whitespace. In the istream library this is part of a greatly expanded collection of state variables and so an extra argument is not provided for the istrstream constructor. However, the obsolete form of istream constructor continues to accept these optional arguments.

## Filebuf

---

Both libraries contain a filebuf class for using streams to do I/O. It is declared in fstream.hpp in the iostream library. The stream library had constructors that implied the use of filebufs. In the iostream library these constructors are replaced by constructors of certain derived classes. The old usage:

```
int fd ;
istream in(fd) ; // file descriptor
ostream out(fd) ; // file descriptor
```

is replaced by:

```
int fd ;
ifstream in(fd) ; // file descriptor
ofstream out(fd) ; // file descriptor
```

The optional extra arguments of the stream constructors (for specifying whitespace skipping and “tying”) are not supported. The equivalent functionality is supported by format state variables.

## Interactions with stdio

---

The libraries differ significantly in the way they interact with stdio. The old stream header stream.hpp included stdio.h and some stream data structures could contain a pointer to a stdio FILE. In the iostream library specialized streams and streambufs (declared in stdiostream.hpp) are provided to make the connection.

The old usage:

```
FILE* stdiofile ;
filebuf fb(stdiofile) ;
istream in(stdiofile) ;
ostream out(stdiofile) ;
```

is replaced by:

```
FILE* stdiofile ;
stdiobuf fb(stdiofile) ;
stdiostream in(stdiofile) ;
stdiostream out(stdiofile) ;
```

In the old library the predefined streams cin, cout, and cerr were directly connected to the stdio FILEs stdin, stdout, and stderr. I/O was mixed character by character. Further, these streams were unbuffered in the sense that insertion and extraction was done by doing character by character puts and gets on the corresponding stdio FILEs. In the iostream library the predefined streams are attached directly to file descriptors rather than to the stdio streams. This means that for output the characters are mixed only as flushes are done and the input buffer of one is not visible to the other.

In practice the biggest problems seem to come from attempts to mix code that uses `stdout` with code that uses `cout`. The best solution is to cause flushes to be inserted whenever the program switches from one library to the other. An alternative is to use:

```
ios::sync_with_stdio() ;
```

This causes the predefined streams to be connected to the corresponding `stdio` files in an unbuffered mode. The major drawback of this solution is the large overheads associated with insertion of characters in this mode. Typically insertion into `cout` is slowed by a factor of 4 after a call of `sync_with_stdio`.

The old stream library contained some stringizing functions that were called with various arguments and returned a string. These are declared in `stream.hpp` and available primarily for compatibility. The only such formatting function that seems to provide a significant functionality that is not easily available in the `iostream` library is `form`, which allows `printf` like formatting. In fact, `form` is just a wrapper for calls to `sprintf`. The programmer can easily write manipulators and inserters that do the same thing.

## Assignment

---

In the old library it was possible to assign one stream to another. This is possible in the `iostream` library only if the left hand side is declared to be an assignable class. A general assignment cannot be allowed because of the interactions of derived classes. What, for example, should be the effect of assigning an `ifstream` to an `istrstream`? Most programs that use this feature can be converted by using a reference or pointer to a stream. The old usage:

```
ostream out ;
out = cout ;
out << x ;
```

can be replaced by:

```
ostream* outp ;
outp = &cout ;
*outp << x ;
```

or:

```
ostream_with_assign out ;
out = cout ;
out << x ;
```

## char insertion operator

---

The stream library did not contain an insertion operator for `char`. So inserting a `char` was taken as inserting an integer value, and it was converted to

decimal. This omission was due to problems with overload resolution in earlier versions of the C++ Language System. Any old code such as:

```
char c ;  
cout << c ;
```

may be replaced by:

```
char c ;  
cout << (int)c ;
```

# ***Stream class directory***

## **header files**

---

The stream classes are defined in the release 2.0 include files `iostream.hpp`, `fstream.hpp` and `strstream.hpp`, containing the basic stream classes, the file classes and the string classes respectively.

In addition the file `iomanip.hpp` defines generic macros which, when invoked, define manipulator classes for any type.

Finally the file `stream.hpp` should be used on its own if you want to continue using the old release 1.2 stream classes.

The detailed contents of the release 2.0 include files are:

### **iostream.hpp**

Contains the base classes `ios` and `streambuf`, the stream classes `istream`, `ostream` and `iostream`, and their variants `istream_withassign`, `ostream_withassign` and `iostream_withassign`.

### **fstream.hpp**

Contains the file stream base classes `filebuf` and `fstreambase`, and the file stream classes `ifstream`, `ofstream` and `fstream`.

### **strstream.hpp**

Contains the string stream base classes `strstreambuf` and `strstreambase`, and the string stream classes `istrstream`, `ostrstream` and `strstream`.

### **stdiostr.hpp**

Contains `stdio` stream base class `stdiobuf` and the stream class `stdiostream`.

### **iomanip.hpp**

Contains generic manipulator class creators, defined in the form of macros to create classes to order.

## **Inheritance structure**

---

The inheritance structure of these classes is as follows:

<b>Class</b>	<b>Derived from</b>
ios	
streambuf	
istream	ios
ostream	ios
iostream	istream ostream
istream_withassign	istream
ostream_withassign	ostream
iostream_withassign	iostream
filebuf	streambuf
fstreambase	ios
ifstream	istream fstreambase
ofstream	ostream fstreambase
fstream	iostream fstreambase
strstreambuf	streambuf
strstreambase	ios
ostrstream	ostream strstreambase
istrstream	istream strstreambase
strstream	iostream strstreambase
stdiobuf	streambuf
stdiostream	ios

The include files also provide the following #defines and typedefs:

### **iostream.hpp**

```
EOF
typedef long streampos;
typedef long streamoff;
```

## class ios -- base class

---

### Public enumerated types

```
enum io_state {
    goodbit    = 0x00,
    eofbit     = 0x01,
    failbit    = 0x02,
    badbit     = 0x04,
    hardfail   = 0x80
};
enum open_mode{
    in         = 0x01,
    out        = 0x02,
    ate        = 0x04,
    app        = 0x08,
    trunc      = 0x10,
    nocreate   = 0x20,
    noreplace  = 0x20,
    binary     = 0x20
};
enum seek_dir {
    beg        = 0,
    cur        = 1,
    end        = 2
};
enum {
    skipws     = 0x0001,
    left       = 0x0002,
    right      = 0x0004,
    internal   = 0x0008,
    dec        = 0x0010,
    oct        = 0x0020,
    hex        = 0x0040,
    showbase   = 0x0080,
    showpoint  = 0x0100,
    uppercase  = 0x0200,
    showpos    = 0x0400,
    scientific
               = 0x0800,
    fixed      = 0x1000,
    unitbuf    = 0x2000,
    stdio      = 0x4000
};
```

### Public data members

```
static const long basefield;
static const long adjustfield;
static const long floatfield;
```

### Public constructors and destructors

```
        ios(streambuf*);
virtual ~ios();
```

Note: ios is also available as a protected constructor

### Public operators

```
        operator void * ();
int      operator      ! ();
```

**Public member functions**

```

long      flags();
long      flags(long);
long      setf(long, long);
long      setf(long);
long      unsetf(long);

int        width();
int        width(int);

char       fill();
char       fill(char);

int        precision();
int        precision(int);
ostream*  tie();
ostream*  tie(ostream*);

int        rdstate();
int        eof();
int        fail();
int        bad();
int        good();
void       clear(int = 0);
streambuf* rdbuf();

static long bitalloc();
static int xalloc();
long&      iword(int);
void*&     pword(int);

static void sync_with_stdio();

int        skip(int);

```

**Protected enumerated types**

```

enum {
    translate = 0x080,
    skipping  = 0x100,
    tied      = 0x200
};

```

**Protected data members**

```

streambuf * bp;
ostream*   x_tie;
int         state;
int         ispecial;
int         ospecial;
long        x_flags;
int         x_precision;
int         x_width;
int         x_fill;
int         isfx_special;
int         osfx_special;
int         delbuf;
int         assign_private;

```

**Protected member functions**

```

        ios();
void     init(streambuf*);
void     setstate(int);
static void (*stdioflush)();

```

## class streambuf -- base class

---

### Constructors and destructors

```

        streambuf();
        streambuf(char *, int);
virtual ~streambuf();

```

### Public member functions

```

virtual streambuf*
        setbuf(char *, int );

virtual streambuf*
        setbuf(char *, int ,int);

int      sgetc();
int      snextc();
int      sbumpc();
void     stoss();
int      sgetn(char *, int);
virtual int do_sgetn(char *, int);
virtual int underflow();
int      sputbackc(char);
virtual int pbackfail(int);
int      in_avail();

int      sputc(int);
virtual int do_sputn(const char *, int);
int      sputn(const char *, int);
virtual int overflow(int = EOF);
int      out_waiting();

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

virtual streampos seekpos(
        streamoff,
        int =(ios::in | ios::out));

virtual int sync();
int      dbp();

```

### Protected member functions

```

char *    base();
char *    ebuf();
int       blen();
char *    pbase();
char *    pptr();
char *    epptr();
char *    eback();
char *    gptr();
char *    egptr();
void      setp(char *, char *);
void      setg(char *, char *, char *);
void      pbump(int);
void      gbump(int);
void      setb(char *, char *, int = 0);
void      unbuffered(int);
int       unbuffered();
int       allocate();
virtual int doallocate();

```

## class istream: -- virtual public ios

---

### Constructors and destructors

```
istream(streambuf*);
istream(streambuf*,
        int,
        ostream* = NULL);
istream(int,
        char *,
        int = 1);
istream(int,
        int = 1,
        ostream* = NULL);

virtual ~istream();
```

Note that there is a protected constructor `istream()`

### Public member operators

```
istream& operator >> ( signed char*);
istream& operator >> (unsigned char*);
istream& operator >> ( signed char&);
istream& operator >> (unsigned char&);
istream& operator >> (short&);
istream& operator >> (int&);
istream& operator >> (long&);
istream& operator >> (unsigned short&);
istream& operator >> (unsigned int&);
istream& operator >> (unsigned long&);
istream& operator >> (float&);
istream& operator >> (double&);
istream& operator >> (long double&);
istream& operator >> (streambuf*);
```

### Generic manipulator operators

```
istream& operator <<
        (istream&
         (* aManipulator)(istream&));

istream& operator <<
        (ios&
         (* aManipulator)(ios&));
```

**Public member functions**

```

int      ipfx(int = 0);
int      ipfx0();
int      ipfx1();
void     isfx();

istream& seekg(streampos);
istream& seekg(streamoff, seek_dir);
streampos tellg();

int      sync();

istream& get(
    signed char*, int,
    char = '\n');
istream& get(
    unsigned char*, int,
    char = '\n');
istream& read(signed char*, int);
istream& read(unsigned char*, int);
istream& getline(
    signed char*, int, char
    = '\n');
istream& getline(
    unsigned char*, int,
    char = '\n');
istream& get(streambuf&, char = '\n');
istream& get(signed char&);
istream& get(unsigned char&);
int      get();

int      peek();
int      gcount();
istream& putback(char);
istream& ignore(int = 1, int = EOF);

```

**Protected member functions**

```

        istream();
int      do_ipfx(int);
void     eatwhite();

```

**class ostream: -- virtual public ios****Constructors and destructors**

```

stream(streambuf *);
ostream(int);
ostream(int, char *);
~ostream();

```

**Public member operators**

```

ostream&    operator << ( signed char);
ostream&    operator << (unsigned char);
ostream&    operator << ( signed short);
ostream&    operator << (unsigned short);
ostream&    operator << ( signed int);
ostream&    operator << (unsigned int);
ostream&    operator << ( signed long);
ostream&    operator << (unsigned long);
ostream&    operator << (float);
ostream&    operator << (double);
ostream&    operator << (long double);
ostream&    operator << (const signed char *);
ostream&    operator << (const unsigned char*);
ostream&    operator << (void *);

ostream&    operator << (streambuf*);

```

**Generic manipulator operators**

```

ostream&    operator <<
              (ostream&
               (* aManipulator)(ostream&));

ostream&    operator <<
              (ios&
               (* aManipulator)(ios&));

```

**Public member functions**

```

int          opfx();
void          osfx();
ostream&     flush();

ostream&     seekp(streampos);
ostream&     seekp(streamoff, seek_dir);
streampos    tellp();

ostream&     put(char);
ostream&     write(const signed char*, int);
ostream&     write(const unsigned char*, int);

```

**Protected member functions**

```

              ostream();
int           do_opfx();
int           do_osfx();

```

**class istream: -- public istream, public ostream****Constructors**

```

              ostream(streambuf*);
virtual ~istream();

```

**Protected constructor**

```

              istream();
class istream_withassign :public istream

```

**Constructors and destructors**

```
        istream_withassign();
    virtual ~istream_withassign();
```

**Public operators**

```
    istream_withassign&
        operator= (istream&);

    istream_withassign&
        operator= (stringstream*);
```

**class ostream\_withassign: -- public ostream**

---

**Constructors**

```
        ostream_withassign();
    virtual ~ostream_withassign();
```

**Public operators**

```
    ostream_withassign&
        operator= (ostream&);

    ostream_withassign&
        operator= (stringstream*);
```

**class istream\_withassign: -- public istream**

---

**Constructors**

```
        istream_withassign();
    virtual ~istream_withassign();
```

**Public operators**

```
    istream_withassign&
        operator= (ios&);

    istream_withassign&
        operator= (stringstream*);
```

**class filebuf: -- public stringstream**

---

**Public data members**

```
    static const int openprot;
```

**Constructors and destructors**

```
    filebuf();
    filebuf(int);
    filebuf(int, char*, int);
    ~filebuf();
```

**Public member functions**

```

int      is_open();
int      fd();

filebuf*  open(
            const char*,
            int,
            int = filebuf::openprot);
filebuf*  close();
filebuf*  attach(int);
virtual int overflow(int = EOF);
virtual int underflow();
virtual int sync();

virtual streampos seekoff(
            streamoff,
            seek_dir,
            int);

virtual streambuf* setbuf(char*, int);

```

**Protected data members**

```

int      xfd;
int      mode;
short    opened;
streampos last_seek;
char*    in_start;
char     lahead[2];

```

**Protected member functions**

```

int      last_op();

```

**class fstreambase: -- virtual public ios**

---

**Constructors and destructors**

```

fstreambase();
fstreambase(
    const char*,
    int,
    int = filebuf::openprot);
fstreambase(int);
fstreambase(int, char*, int);
fstreambase();

```

**Public member functions**

```

void      open(
            const char*,
            int,
            int = filebuf::openprot);
void      attach(int);
void      close();
void      setbuf(char*, int);
filebuf*  rdbuf();

```

---

**class ifstream: -- public fstreambase, public istream**

---

**Constructors and destructors**

```
ifstream();  
ifstream(  
    const char*,  
    int = ios::in  
    int = filebuf::openprot);  
ifstream(int);  
ifstream(int, char*, int);  
~ifstream();
```

**Public member functions**

```
filebuf*    rdbuf();  
void        open(  
    const char*,  
    int= ios::in,  
    int = filebuf::openprot);
```

---

**class ofstream: -- public fstreambase, public ostream**

---

**Constructors**

```
ofstream();  
ofstream(  
    const char*,  
    int = ios::out,  
    int = filebuf::openprot);  
ofstream(int);  
ofstream(int, char*, int);  
~ofstream();
```

**Public member functions**

```
filebuf*    rdbuf();  
void        open(  
    const char*,  
    int= ios::out,  
    int = filebuf::openprot);
```

---

**class fstream: -- public fstreambase, public iostream**

---

**Constructors and destructors**

```
fstream();  
fstream(  
    const char*,  
    int,  
    int = filebuf::openprot);  
fstream(int);  
fstream(int , char*, int);  
~fstream();
```

**Public member functions**

```

filebuf*   rdbuf();
void       open(
            const char *,
            int,
            int = filebuf::openprot);

```

**class stringstream: -- public streambuf**

---

**Constructors and Destructors**

```

stringstream();
stringstream(int);
stringstream(
    void * (*a)(long),
    void (*f)(void*));
stringstream(
    signed char *_s,
    int,
    signed char *_start=NULL);
~stringstream();

```

**Public member functions**

```

void       freeze(int = 1);
char *     str();
virtual int doallocate();
virtual int overflow(int);
virtual int underflow();
virtual streambuf
            *setbuf(char *, int);
virtual streampos
            seekoff(
                streamoff, seek_dir,
                int);

```

**class stringstreambase: -- public virtual ios**

---

**Public constructor**

```
stringstream* rdbuf();
```

**Protected constructors**

```

stringstreambase(char *,
                  int,
                  char *);
stringstreambase();
~stringstreambase();

```

**class istrstream: -- public stringstreambase, public istream**

---

**Constructors**

```

istrstream(char *);
istrstream(char *, int);
~istrstream();

```

## **class ostream: -- public ostreambase, public ostream**

---

### **Constructors**

```
ostream();  
ostream(char *, int, int);  
~ostream();
```

### **Public member functions**

```
char *    str();  
int       pcount();
```

## **class stringstream: -- public ostreambase, public ostream**

---

### **Constructors**

```
stringstream();  
stringstream(char *,  
              int _sz,  
              int _m);  
~stringstream();
```

### **Public member functions**

```
char *    str();
```

# ***Stream class reference section***

## **Conventions**

---

Class members are documented in logical rather than lexical order. Readers who want to look up a particular member should use the member index to locate it and then read the relevant reference section.

Member functions and manipulators are used as headings with an example of their use, rather than a declaration, since the formal declaration is given in the class index.

Classes are in hierarchical order; within each class, members are grouped functionally, on the basis of the purpose they serve.

Identifiers convey their class where possible, so `aStreambuf` stands for an instance of class `streambuf`. The identifiers `in` and `out` are used for instances of classes `istream` and `ostream` respectively. Note that most `ios` services are intended for use by derived classes, so that examples of usage may refer to use by a derived class rather than an `ios` instance.

## **class ios**

---

Class `ios` is the base class for all stream classes, all of which inherit from `ios`. An `ios` object provides format and error handling. It uses a `streambuf` protected member, which is always associated with it, to handle the actual supply and disposal of information. It can be thought of as a user-friendly and error-preventive envelope for the simple transport of data provided by `streambuf`s; when efficiency is the overriding requirement it is possible to use `streambuf` objects directly.

It is used by all derived stream classes as a virtual base class, so that its members can only be inherited once. Applications usually create instances of classes derived from `ios`.

Data and function members declared in `ios` are those common to both input and output streams. They deal with:

- Error and condition handling;
- Data formatting;
- File creation, opening and closing, and positioning.

In addition to the `streambuf` member which actually handles input and output, class `ios` maintains:

- A set of format state variables, with associated operators and functions, used by derived classes to control the translation of objects to and from their text representation;
- An error state variable with a set of state operators and functions through which the application can set and detect stream conditions arising from external conditions;
- An assortment of state constants, implemented as enumerations, which define the range of possible values the state variables may have and are available to the application to interact with streams. Because they are defined inside class `ios`, the names of these constants will not conflict with names in the application program

The state variables behave like an array of independent fields, though most are in fact implemented as a bit vector in which each bit represents a Boolean condition;

### **Constructors, initialization and assignment.**

As a class that is usually inherited as a virtual class, the `ios` constructors require some attention. Facilities are provided for initializing the `ios` members of a derived class instance independently of the `ios` constructor.

```
ios anIos(aStreambuf);
aStreambuf becomes the stream buffer associated with anIos. The result is not defined if aStreambuf is null.
ios I();
ios::init(aStreambuf);
ios() is the default constructor which will be invoked if a derived class constructor does not explicitly invoke either of the above. It does no work. The actual initialization is carried out by the member function init(aStreambuf), which is available as a protected member for use by derived classes.
```

The preferred way to set up a constructor for a derived class is therefore similar to the following:

```
class D: virtual public ios{
    D(streambuf& aStreambuf){
        /* ... some D initialization code ... */
        init(aStreambuf);
        /* ... some more D initialization code ... */
    }
}
```

This is more general than the following:

```
class D: virtual public ios{
    D(streambuf& aStreambuf):ios(aStreambuf){
        /* ... all D initialization code */
    }
}
```

In the second case, because `D` is derived from `ios`, the `ios` constructor will be invoked before `D`'s initialization code. Therefore, if `D` wants to do any processing beforehand, the first form is necessary.

```
ios I(aStream);
anIos = anotherIos;
```

Assignment of instances of class `ios` is not well defined in general. The copy constructor and the assignment operator are private and the compiler will fault attempts to copy `ios` objects. Special classes (`istream_withassign` etc) are defined for stream assignment; they arrange to copy pointers to streams rather than the streams themselves.

### **Data member access**

Apart from the format and error state variables defined below, class `ios` provides the access function `rdbuf`:

```
aStreambuf = anIos->rdbuf()
```

This yields the stream buffer member of `ios`.

### **Format control**

`ios` instances maintain a format flag variable and a series of other format control variables which determine the relation between binary data and its representation as strings of characters. They have no other effect on the transport of data and may be set independent of all other operations.

The flags are actually held as a private long integer in which specific bitfields are reserved for specific format control functions. An enumeration provided by `ios` and detailed below allows applications to refer to these bitfields by convenient mnemonic names.

The most general member functions dealing with the format flags are `flags`, `setf` and `unsetf` (see below), which in principle allow the program set any combination of bits in the flag variable. Other format variables, such as the field width, are accessed by member functions.

More specialized functions and manipulators are provided that put the flags into a well-defined state. For example the manipulators `dec`, `oct` and `hex` control the conversion base, and the precision member function sets the floating point precision.

Unfortunately the format state is not fully encapsulated. Not all fields can be set using special-purpose functions. Some - for example the `adjustfield` bits — can only be accessed via `setf`.

**Caution:** Because the format state variable is directly accessed by `setf` you can set mutually contradictory flags. To guarantee a well-defined result we recommend using member functions such as `dec` where they exist, and where they do not to follow the usage examples given below.

### **Format flag general purpose access functions**

The following functions and manipulators all access the format flag word directly and set bits in it as defined by the format flag enumeration defined below. Each returns the value which the flag word had before the operation

was performed; where a bit field is specified, only the bits it contains are returned.

```
f = anIos.flags();
f = anIos.flags(aFlagWord);
flags simply returns the current flags without changing them. flags(aFlagWord) sets the flag word to be aFlagWord.
f = anIos.setf(bitValues);
in << setiosflags(bitValues);
setf(bitValues) and setiosflags(bitValues) turn on all the bits which are set in bitValues, without changing any others.
f = anIos.setf(aBitValue, aBitField);
in << resetiosflags(aBitValue, aBitField);
setf(aBitValue, aBitField) and resetiosflags(aBitValue, aBitField) turn on the bits specified by aBitValue, but turn off all other bits that fall within aBitField.
```

Thus, for example:

```
anIos.setf(ios::hex, ios::basefield)
```

will set the basefield bits of the flag word to be hex. This is the preferred technique for setting values into a bitfield of more than one bit. The wrong effect would be produced by

```
anIos.setf(ios::hex); //wrong
```

because if the dec bit was previously set, the above statement would not unset it.

```
f = anIos.unsetf(aBitField);
unsetf(aBitField) unsets the bits set in b.
```

## **Special purpose access functions**

Each section below deals with one format control operation, and shows how a specific format variable, or a specific flag, can be changed. Where a member function or manipulator is defined these are given first. If the operation is controlled by a bitfield in the flag variable, the preferred usage for setting and unsetting it is given.

```
old_i = width(i);
in >> setw(i);
```

A format variable controls field width and may be set with the `width` member function, which sets the width to `i` and returns the former width. Several format states refer to this for their effect.

For output, when the width is zero — the default — then the minimum number of characters needed to represent an object will be used. When it is non-zero it is interpreted as a minimum width. The next object to be output is then padded with a fill character — set by the fill member function — to width characters. The stream classes never truncate on output, and there is no way to specify a maximum field size.

On input the field width is taken to be the number of characters expected for the object to be read.

The width is always reset to its default value of 0 after each read, so that a call to width behaves as if it were setting a parameter to the next operation. As a precaution against looping, when the width field is 0 and whitespace-skipping turned off, an input operator reading a scalar type will return an error unless it finds what it is looking for.

```
f = fill(char fillChar);
in >> setfill(i);
out << setfill(i);
```

sets the fill character to be fillchar and returns the previous value of this character. The default is a space. The position of the fill character is dealt with by the adjustfield flag (see below).

### **The format flag word**

An anonymous enumerated type establishes a list of conditions which can be set in the state vector. The implementation permits the user to set any bit independently of each other. The bits are grouped as follows:

### **Padding and field adjustment**

A protected data member adjustfield controls padding.

The field adjust state can have one of three mutually exclusive values contained in the adjustfield bits.

```
anIos.setf(ios::left, ios::adjustfield);
anIos.setf(ios::right, ios::adjustfield);
anIos.setf(ios::internal, ios::adjustfield);
```

When left is set, fill characters are added at the end of a field so that the representation is left-adjusted. With right set the converse is true. With internal set the fill character is inserted between the value and any leading sign or base. If no flags are set, right padding is done.

If both left and right flags are set right takes precedence.

### **Conversion**

A protected bitfield basefield contains the bits which deal with the conversion base for integer types

```
out >> dec;
in << dec;
anIos.setf(ios::dec, ios::adjustfield);
out >> oct;
in << oct;
anIos.setf(ios::oct, ios::adjustfield);
out >> hex;
in << hex;
anIos.setf(ios::hex, ios::adjustfield);
```

These control the conversion base for integer types.

If dec is set the conversion base is 10; if oct it is octal and if hex it is hexadecimal. If none is set output is decimal but input follows C++ lexical conventions, so that for example

```
0xFF
```

will be read as hexadecimal FF.

```
anIos.setf(ios::showbase);
```

If the showbase bit is set, output will follow C++ lexical conventions. Therefore, for example, if hex is set the above would be output as 0xFF; without showbase it would be output as FF. By default showbase is not set.

```
anIos.setf(ios::showpos);
```

If showpos is set a '+' will be inserted into the decimal conversion of a positive integer value.

```
anIos.setf(ios::showpoint);
```

If showpoint is set, trailing zeroes and decimal points appear in the result of a floating point conversion.

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

If uppercase is set then an uppercase X will be used for hexadecimal output (if showbase is set) and uppercase E will be used for floating point numbers in scientific notation.

## **Floating point conversion**

A protected bitfield floatfield controls floating point format. It can have one of two mutually exclusive values:

```
anIos.setf(ios::scientific);
```

If scientific is set conversion uses scientific notation in which there is one digit before the decimal point, and the number of digits after is equal to the precision, which is set using the precision member function defined below. The presentation is also governed by the uppercase and showpoint flags defined above.

```
anIos.setf(ios::fixed);
```

If fixed is set, values are converted to decimal notation with precision digits after the decimal point.

If neither are set values will be output depending on their magnitude. Scientific notation is used if the exponent is less than -4, or greater than the precision.

```
j=anIos.precision(i)
```

Precision determines the number of digits output after the decimal point. The default value is 6. The function returns the previous setting.

## **Whitespace skipping**

```
anIos.setf(ios::skipws);
```

If and only if this is set, whitespace is skipped during input of all language-defined scalar types. If skipping is not set and whitespace is encountered while reading a scalar type, an error will be signalled unless the width member function is non-zero.

## **Buffer control**

Two further flags in the format word control the synchronization of stream output with the operating system and with other streams. They are not strictly speaking format flags, but are included in the format word for completeness.

```
anIos.setf(unitbuf);
```

See also `ostream::osfx()`

After this statement an output buffer will be flushed after the input or output of each object.

```
anIos.setf(stdio)
```

See also: `ios::sync_with_stdio`, `ostream::osfx`.

If `stdio` is set, output is synchronized with the old C standard i/o streams. It is only necessary if you are combining stream library output with C input-output calls such as `printf`. It imposes a substantial time penalty.

## **User-defined format flags**

Several member functions help create derived classes than need additional format flags.

```
aLong = ios::bitalloc()
```

returns a long with a single, previously unallocated bit set. This gives an extra flag to users who need one. This flag can then be passed to `ios::setf`.

```
anInt = anIos.xalloc()
```

This returns a new — unused — index into an array of words that can be used as format variables by derived classes.

```
aLong = anIos.iword(i)
```

When `xalloc` has been used to allocate an index, this will return a reference to the *i*'th user-defined word. Returns the same value as `pword` below but with a different type cast.

```
aVoidPointer = anIos.pword(i)
```

If `xalloc` has allocated *i*, this returns a reference to the *i*'th user-defined word. `pword` returns the same as `iword` but with a different type cast.

## Error handling

An ios instance has an internal error state which is handled much in the same way as the format state. The enumeration `io_states` lists the five error conditions which may hold.

These conditions all apply independent of each other and any combination of error states is in principle possible.

The bits are defined as follows:

```
goodbit   = 0x00
eof        = 0x11
failbit    = 0x02
badbit     = 0x04
hardfail   = 0x80
```

**goodbit** is simply a pseudonym for no bits set at all, which means no errors are being signalled. **eofbit** is set when end of file has been reached. **badbit** signifies a severe error from which recovery is usually impossible. **hardfail** has the same meaning but signifies some hardware fault has probably caused the error. **failbit** is a user-controlled bit which is used by convention to signify that some operation has failed but that the stream is still usable once the condition has been dealt with.

The following member functions report on or modify the error state:

```
i = anIos.rdstate()
```

yields the current value of the error state.

```
anIos.clear(anErrorState)
```

Sets the error state to be `anErrorState`. Note that the convention here is different from the `setf` method, and is in fact the same as the `flags` method for setting the format state. To set a bit without clearing previously set bits the usage is:

```
anIos.clear(ios::failbit | anIos.rdstate());
i = anIos.good()
```

Sets `i` to zero if error bits are set, non-zero otherwise.

```
i = anIos.eof()
```

Sets `i` non-zero if `eofbit` is set - conventionally, if an end of file has been encountered. If `eofbit` is not set, `i` will become zero.

```
i = anIos.fail()
```

Sets `i` non-zero if either `badbit`, `hardfail` or `failbit` is set, zero if none of them is set. It may be possible to continue to continue reading after such a condition if the program can establish and correct the cause.

```
i = anIos.bad()
```

Sets `i` non-zero if `badbit` is set, zero otherwise. Usually indicates an unrecoverable error caused by a condition arising outside the application itself.

## **Special error operators**

ios provides two special operators to check error states, both of which are really provided for the notational brevity so beloved of C programmers.

```
ios::operator void*()
```

Returns the ios address if no error condition holds, and zero otherwise. This permits expressions like:

```
if (in)...;  
if (in >>)...;
```

which will implement the if-clause provided nothing has gone wrong.

```
ios::operator !()
```

Returns non-zero if either failbit, hardfail or failbit is in the error state, permitting expressions like

```
if(!out)...;
```

## **Flushing and synchronization**

```
ios::sync_with_stdio()  
oldOstream = anIos.tie(newOstream);
```

This ‘ties’ the ios to newOstream and if it was previously tied, yields the former tied stream in oldOstream. This provides for ios instances to be flushed automatically. If anIos, which is usually an istream, needs more characters after the above statement, newOstream will be flushed. Thus if newOstream was the screen and anIos the keyboard, this would ensure that prompts were put up on the screen before the user was expected to reply to them.

```
anOstream = anIos.tie();
```

Yields the ostream that anIos is tied to, or null if there is no tie.

## **Stream positioning**

Although class ios contains the enumerated type seek\_dir, it is used by the streambuf class and is therefore described there. File position functions are defined for derived classes, where they should likewise be referenced.

## **File open mode**

Although class ios contains the enumerated type open\_mode, this is used by the filebuf class where it is described

---

## **class istream**

## class `istream_withassign`

---

Class `istream` is derived directly from `ios` and is the main input (extraction) class. It adds standard input (`>>`) operators for all the main C++ base types, and a number of specialized character-handling procedures such as `get` with which user-defined extraction operators can be built.

`istream` instances cannot be assigned to. A special class, `istream_withassign` is derived from `istream` for cases where stream input needs to be redirected. It inherits all `istream` methods but is slightly less efficient.

### Constructors and initialization

```
istream in(aStreambuf);
```

Initializes the state variables inherited from `ios` and associates `aStreambuf` with the `istream`.

```
istream_withassign();
```

Does no initialization: this has to be done by invoking one of the two initialization methods which follow. Either of these provide an `AssignableIstream` with a `streambuf` and initialize it ready for use.

```
anAssignableIstream = aStreambuf;
```

Sets the stream buffer used by an `AssignableIstream` to be `aStreambuf` and initializes all the former's state variables.

```
anAssignableIstream = in;
```

Associates the `streambuf` member of `in` with an `AssignableIstream` and initializes all the latter's state variables.

### Input prefix function

#### **`i = in.ipfx(need);`**

See also `ios::tie`, `ios::sync_with_stdio` and `ios` error-handling.

The input prefix function is called by all input primitives to prepare their stream for input. It reports on the error state and carries out any necessary flushing operations.

If `in` has a non-zero error state, this returns zero immediately. If necessary, and if it is non-null, any `ios` tied to `in` is flushed. Flushing is considered necessary if either the `need` parameter is 0 or if there are fewer than `need` characters immediately available. If `ios::skipws` is set and `need` is zero, then leading whitespace characters are extracted from `in`. `ipfx` returns zero if an error occurs while skipping whitespace; otherwise it returns non-zero.

Formatted input functions call `ipfx(0)`, while unformatted input functions call `ipfx(1)`; see below.

## **Formatted input**

```
in>>aSimpleType
```

See also: ios format conventions.

The >> operators for the main predefined language types are defined in the istream class.

An >> operator performs the follows actions in turn:

- It calls ipfx(0).
- If this returns non-zero, no further action is taken, so the next input operation will yield the same result.
- If ipfx(0) returns zero, characters are extracted from in and converted according to the type of x and the state of the format flags. The converted value is stored in x.
- Any problems are indicated by setting the error state of istream in. ios::failbit means that the input was not a representation of the required type. ios::badbit indicates that attempts to extract characters failed.

ios::hardfail indicates that a hardware error occurred while attempting to extract characters.

- istream in is always returned.

For all primitive types >> member operators have been defined so that the application does not have to reference or dereference the object. The single format

```
in >> x;
```

is valid for all types including char\*, which is interpreted as a pointer to a string. This is achieved for the non-pointer types by specifying a reference-type parameter for the operator.

Because << returns a reference to in, successive invocations can be stacked as in:

```
in >> x >> y >> z;
```

If these conventions are maintained for user-defined types, these operations can be used with the same syntax and condition-handling features as the member functions described here.

## **Effects of >> for the primitive types**

```
char*, unsigned char*
```

Characters are stored in the array pointed at by x until a whitespace character is found in istream in. The terminating whitespace is left in. If in.width is non-zero it is taken to be the size of the array, and no more than in.width()-1 characters are extracted. A terminating null character (0) is always stored

(even when nothing else is done because of in's error status). in.width is reset to 0.

```
char&, unsigned char&
```

A character is extracted and stored in x.

```
short&, unsigned short&,
int&, unsigned int&,
long&, unsigned long&
```

Characters are extracted and converted to an integral value according to the conversion specified in in's format flags. Converted characters are stored in x. The first character may be a sign (+ or -). After that, if ios::oct, ios::dec, or ios::hex is set in in.flags, the conversion is octal, decimal, or hexadecimal respectively. Conversion is terminated by the first non-digit, which is left in. Octal digits are the characters '0' to '7'. Decimal digits are the octal digits plus '8' and '9'. Hexadecimal digits are the decimal digits plus the letters 'a' through 'f' (in either upper or lower case). If none of the conversion base format flags is set, then the number is interpreted according to C++ lexical conventions. That is, if the first characters (after the optional sign) are 0x or 0X a hexadecimal conversion is performed on following hexadecimal digits. Otherwise, if the first character is a 0, an octal conversion is performed, and in all other cases a decimal conversion is performed. ios::failbit is set if there are no digits (not counting the 0 in 0x or 0X during hex conversion) available.

```
float&, double&
```

Converts the characters according to C++ syntax for a float or double, and stores the result in x. ios::failbit is set if there are no digits available in istream in or if it does not begin with a well formed floating point number.

### **Streambuf input operator**

```
in>>aStreambuf
```

If ios.ipfx(0) returns non-zero, reads characters from ios's streambuf and inserts them into aStreambuf. Input stops when EOF is reached. Always returns in.

### **Unformatted input functions**

These functions call ipfx(1) and proceed only if it returns non-zero:

```
inp=&in.get(ptr,len,delim)
```

Extracts characters and stores them in the byte array beginning at ptr and extending for len bytes. Extraction stops when delim is encountered (delim is left in and not stored), when in has no more characters, or when the array has only one byte left. get always stores a terminating null, even if it doesn't extract any characters from in because of its error status. ios::failbit is set only if get encounters an end of file before it stores any characters.

```
inp=&in.get(c)
```

Extracts a single character and stores it in c.

```
inp=&in.get(sb,delim)
```

Extracts characters from `in.rdbuf` and stores them into `sb`. It stops if it encounters end of file or if a store into `sb` fails or if it encounters `delim` (which it leaves in istream `in`). `ios::failbit` is set if it stops because the store into `sb` fails.

```
i=in.get()
```

Extracts a character and returns it. `i` is EOF if extraction encounters end of file. `ios::failbit` is never set.

```
inp=&in.getline(ptr,len,delim)
```

Does the same thing as `in.get(ptr,len,delim)` with the exception that it extracts a terminating `delim` character from `in`. In case `delim` occurs when exactly `len` characters have been extracted, termination is treated as being due to the array being filled, and thus `delim` is left in.

```
inp=&in.ignore(n,d)
```

Extracts and throws away up to `n` characters. Extraction stops prematurely if `d` is extracted or end of file is reached. If `d` is EOF it can never cause termination.

```
inp=&in.read(ptr,n)
```

Extracts `n` characters and stores them in the array beginning at `ptr`. If end of file is reached before `n` characters have been extracted, `read` stores whatever it can extract and sets `ios::failbit`. The number of characters extracted can be determined via `in.gcount`.

Other members are:

```
i=in.gcount()
```

Returns the number of characters extracted by the last unformatted input function. Formatted input functions may call unformatted input functions and thereby reset this number.

```
i=in.peek()
```

Begins by calling `in.ipfx(1)`. If that call returns zero or if `in` is at end of file, it returns EOF. Otherwise it returns the next character without extracting it.

```
inp=&in.putback(c)
```

Attempts to back up `in.rdbuf`. `c` must be the character before `in.rdbuf`'s get pointer. (Unless other activity is modifying `in.rdbuf` this is the last character extracted from `in`.) If it is not, the effect is undefined. `putback` may fail (and set the error state). Although it is a member of istream, `putback` never extracts characters, so it does not call `ipfx`. It will, however, return without doing anything if the error state is non-zero.

```
i=&in.sync()
```

Establishes consistency between internal data structures and the external source of characters. Calls `in.rdbuf()->sync()`, which is a virtual function, so the details depend on the derived class. Returns EOF to indicate errors.

```
in>>manip
```

Equivalent to `manip(in)`. Syntactically this looks like an extractor operation, but semantically it does an arbitrary operation rather than converting a sequence of characters and storing the result in `manip`. A predefined manipulator, `ws`, is described below.

Member functions related to positioning:

```
inp=&in.seekg(off,dir)
```

Repositions `in.rdbuf`'s get pointer. See introduction for a discussion of positioning.

```
inp=&in.seekg(pos)
```

Repositions `in.rdbuf`'s get pointer. See introduction for a discussion of positioning.

```
pos=in.tellg()
```

The current position of `ios.rdbuf`'s get pointer. See introduction for a discussion of positioning.

## **Manipulators**

```
in>>ws
```

Extracts whitespace characters.

```
in>>dec
```

Sets the conversion base format flag to 10.

```
in>>hex
```

Sets the conversion base format flag to 16.

```
in>>oct
```

Sets the conversion base format flag to 8.

## **istream global variables**

```
cin << x;
```

The stream library supplies a standard input stream, `cin`. This is actually an `istream_withassign`, so it can be reassigned. Thus

```
cin = in;
```

will redirect all console input to the stream `in`.

**Warning: There is no overflow detection on conversion of integers.**

## ostream - formatted and unformatted output

---

```
ostream&    flush();
istream&    operator=(istream&);
istream&    operator=(streambuf*);
extern ostream_withassign cout;
extern ostream_withassign cerr;
extern ostream_withassign clog;
```

Class `ostream` is derived directly from `ios`. It is the main output (insertion) class. It adds standard output (<<) operators for the standard base types, and specialized character-handling procedures such as `put`, with which user-defined insertion operators can be built.

**ostream instances cannot be assigned to.** A special class, `ostream_withassign` is derived from `ostream` for cases where stream input needs to be redirected. It inherits all `ostream` methods but is slightly less efficient.

### Constructors and assignment

```
ostream out(aStreambuf)
```

Initializes the state variables inherited from `ios` and associates `aStreambuf` with `out`.

```
ostream_withassign assignableOstream();
```

Does no initialization: this has to be done by invoking one of the two initialization methods which follow. Either of these provide the `anAssignableOstream` with a `streambuf` and initialize it ready for use.

```
anAssignableOstream = aStreambuf;
```

Sets the stream buffer used by `assignableOstream` to be `aStreambuf` and initializes all the former's state variables.

```
anAssignableOstream = out;
```

Associates the `streambuf` member of `out` with `anAssignableOstream` and initializes all the latter's state variables.

### Input prefix function

```
i = out.opfx();
```

See also `ios::tie`, `ios::sync_with_stdio` and `ios` error-handling.

The output prefix function is called by all output primitives to prepare for output. It reports on the error state and carries out any necessary flushing operations.

If `out` has a non-zero error state, `opfx` returns zero immediately. If `anOtherOstream` is tied to `out` then `anOtherOstream` is flushed. `opfx` returns zero if the stream was in an error state when it was called.

### Output suffix function

```
out.osfx();
```

This performs standard buffer flushing actions after output has been processed and before returning from inserters:

- If `ios::unitbuf` is set, `osfx` flushes out.
- If `ios::stdio` is set, `osfx` flushes `stdout` and `stderr`.

The `unitbuf` flush ensures that objects are sent to the output device one at a time, instead of waiting until a complete buffer is full. It is a useful compromise between fully buffered output and single-character output. Single-character output is rather inefficient because it makes a system call for each character written, while fully buffered output is unsuited to interactive devices because the output is not seen at its point of generation.

The `stdio` flush ensures synchronization with the older C-based input-output layer if this is present.

`osfx` is called by all predefined output operations. It should be called by user-defined output operators after any direct manipulation of the `streambuf`, though a user-defined operator which simply calls predefined operators will automatically invoke `osfx`.

`osfx` is not called by the binary output functions.

## Output functions (inserters)

### Formatted output

```
outs << anObject;
```

See also: `ios` format conventions.

The standard output sequence is as follows:

- `outs.opfx` is called. If it returns zero, nothing is done.
- Otherwise the character representation of `x` is inserted into `outs.rdbuf()`.
- If an error arises during output, output ceases and the error state of `out` is set.
- The operator always returns a reference to `out`.

`x` is converted according to rules that depend on the type of `x` and the format state of `out`. Inserters are defined for the following types, with conversion rules as described below:

```
out << aString
```

Outputs the sequence of characters up to but not including the terminating null of the string `x` points at.

```
out << any integral type except char and unsigned char
```

If `x` is positive, the representation contains a sequence of decimal, octal, or hexadecimal digits with no leading zeros according to whether `ios::dec`, `ios::oct`, or `ios::hex`, respectively is set. If none of those flags are set, conversion defaults to decimal. If `x` is zero, the representation is a single zero

character (0). If *x* is negative, a decimal conversion outputs a minus sign (-) followed by decimal digits. If *x* is positive and `ios::showpos` is set, decimal conversion converts it to a plus sign (+) followed by decimal digits. The other conversions treat all values as unsigned. If `ios::showbase` is set in the `ios` format flags, the hexadecimal representation contains 0x before the hexadecimal digits, or 0X if `ios::uppercase` is set. If `ios::showbase` is set, the octal representation contains a leading 0.

```
out << aVoidPointer;
```

Pointers are converted to integral values and then converted to hexadecimal numbers as if `ios::showbase` were set.

```
out << aFloat;
out << aDouble;
```

The arguments are converted according to the current values of `out.precision()`, `out.width()` and the `out` format flags `ios::scientific`, `ios::fixed`, and `ios::uppercase`. The default value for `outs.precision()` is 6. If neither `ios::scientific` nor `ios::fixed` is set, either fixed or scientific notation is chosen for the representation, depending on the value of *x*.

```
out << aChar;
out << anUnsignedChar
```

No special conversion is necessary.

After the representation is determined, padding occurs. If `outs.width()` is greater than 0 and the representation contains fewer than `outs.width()` characters, then enough `outs.fill()` characters are added to bring the total number of characters to `ios.width()`. If `ios::left` is set in the `ios` format flags, the sequence is left-adjusted, that is, characters are added after the characters determined above. If `ios::right` is set, the padding is added before the characters determined above. If `ios::internal` is set, the padding is added after any leading sign or base indication and before the characters that represent the value. `ios.width()` is reset to 0, but all other format variables are unchanged. The resulting sequence is inserted into the `streambuf` member of `out`.

```
out << aStreambuf
```

If `out.opfx` returns non-zero, the sequence of characters that can be fetched from `aStreambuf` are inserted into `out.rdbuf()`. Insertion stops when no more characters can be fetched from `sb`. No padding is performed. Always returns `out`.

## **Unformatted output**

```
out.put(c);
```

Inserts *c* into the `out.rdbuf()` (the `streambuf` member of `out`.) Sets the error state if the insertion fails.

```
outs.write(aString,n);
```

Inserts the *n* characters starting at `aString` into `out.rdbuf()`. These characters may include zeros — i.e. `aString` need not be a null terminated string.

## **Buffer flushing**

See also `ostream::osfx`; stream class introduction.

```
out.flush();
```

Storing characters into a `streambuf` does not always cause them to be consumed (e.g. written to the external file) immediately. `flush` causes any characters that may have been stored but not yet consumed to be consumed by calling `out.rdbuf()->sync`.

## **Positioning functions**

See also: `streambuf::seekp`; stream class introduction.

```
out.seekp(aStreamoff, aSeek_direction);
```

Repositions the `outs.rdbuf()` put pointer to the position specified by `aStreamoff` and `aSeek_direction`.

This operator returns the modified `out` as an `ostream` reference.

```
out.seekp(aStreamPos)
```

Repositions the `outs.rdbuf()` put pointer.

This operator returns the modified `out` as an `ostream` reference.

```
aStreamPos = outs.tellp()
```

Returns the current position of the `outs.rdbuf()` put pointer.

## **Manipulators**

The `ostream` class has two generic manipulator `<<` operators declared thus:

```
ostream& ostream::operator <<
    (ios& (*aManipulator)(ios &i))
ostream& ostream::operator <<
    (ostream& (*aManipulator)
    (ostream &anOstream))
```

The first takes as parameter an object which looks like an ordinary data object but is in fact a function with an `ios` parameter returning an `ios`. The second is the same, except that the parameter and return value are specialized to `ostreams`.

This makes it possible to define manipulators for the `ostream` class as follows:

```
ios& aManipulator (ios&)
{...}
ostream& aManipulator (ostream&)
{...}
```

After either declaration `aManipulator` will be invoked when the following statement is executed:

```
out << aManipulator;
ostream manipulators, just like istream manipulators, are therefore all declared and defined as
functions taking an ios& parameter and returning an ios&. In common usage, however, they are stacked
as part of a sequence of << calls along with normal data objects. Their definition is as follows:
out << aManipulator;
aManipulator(out);
```

The ostream manipulators are declared as ios functions taking an ios parameter. The above two statements are therefore equivalent.

Applications may define their own manipulators as explained in the section on manipulators below. Predefined manipulators are as follows:

```
out<<endl
```

Ends a line by inserting a newline character and flushing.

```
out<<ends
```

Ends a string by inserting a null character.

```
out<<flush
```

Flushes out.

```
out<<dec
```

See also ios::dec.

Sets the conversion base format flag to 10.

```
out<<hex
```

See also ios::hex.

Sets the conversion base format flag to 16.

```
out<<oct
```

Sets the conversion base format flag to 8.

## class streambuf

---

### Public interface

#### Elementary output

```
c=sb->sputc()
```

Moves the get pointer forward one character and returns the character it moved past. Returns EOF if the get pointer is currently at the end of the sequence.

```
c=sb->snextc()
```

Moves the get pointer forward one character and returns the character following the new position. It returns EOF if the pointer is currently at the end of the sequence or is at the end of the sequence after moving forward.

```
c=sb->sgetc()
```

Returns the character after the get pointer. Contrary to what most people expect from the name IT DOES NOT MOVE THE GET POINTER. Returns EOF if there is no character available.

```
sb->stoss()
```

Moves the get pointer forward one character. If the pointer started at the end of the sequence this function has no effect.

```
i=sb->sputback(c)
```

Moves the get pointer back one character. *c* must be the current content of the sequence just before the get pointer. The underlying mechanism may simply back up the get pointer or may rearrange its internal data structures so the *c* is saved. Thus the effect of *sputback* is undefined if *c* is not the character before the get pointer. *sputback* returns EOF when it fails. The conditions under which it can fail depend on the details of the derived class.

```
i=sb->sgetn(ptr,n)
```

Fetches the *n* characters following the get pointer and copies them to the area starting at *ptr*. When there are fewer than *n* characters left before the end of the sequence *sgetn* fetches whatever characters remain. *sgetn* repositions the get pointer following the fetched characters and returns the number of characters fetched.

### **Elementary input functions**

```
i=sb->sputc(c)
```

Stores *c* after the put pointer, and moves the put pointer past the stored character; usually this extends the sequence. It returns EOF when an error occurs. The conditions that can cause errors depends on the derived class.

```
i=sb->sputn(ptr,n)
```

Stores the *n* characters starting at *ptr* after the put pointer and moves the put pointer past them. *sputn* returns the number of characters stored successfully. Normally this is *n*, but it may be less when errors occur.

### **Position functions**

An instance of class *streambuf* can be positioned either absolutely or relatively. Position information is stored in one of two special types reserved for relative and absolute positions. When specifying a relative stream position the caller must also specify whether it is relative to the beginning of the file, the current position in the file, or the end of the file. An enumerated type *seek\_dir*, defined in class *ios*, specifies these three options.

```
enum seek_dir
enum seek_dir
{ beg,
  cur,
  end
}
```

Class `streambuf` defines two types which are used for position information.

```
typedef long streamoff, streampos;
```

Class `streambuf` supplies these two typedefs which are used for stream offsets and absolute positions respectively. The programmer is advised to use this type explicitly since in future implementations the definition may change.

```
pos=sb->seekoff(off,dir,mode)
```

Repositions the get and/or put pointers. `mode` specifies whether the put pointer or the get pointer is to be modified. Both bits may be set in which case both pointers should be affected.

```
ios::in
```

get pointer modified

```
ios::out
```

put pointer modified

`off` is interpreted as a byte offset. (Notice that it is a signed quantity.) The meanings of possible values of `dir` are:

```
ios::beg
```

The beginning of the stream.

```
ios::cur
```

The current position.

```
ios::end
```

The end of the stream (end of file).

Not all classes derived from `streambuf` support repositioning. `seekoff` will return EOF if the class does not support repositioning. If the class does support repositioning, `seekoff` will return the new position or EOF on error.

```
pos=sb->seekpos(pos,mode)
```

Repositions the `streambuf` get and/or put pointer to `pos`. `mode` specifies which pointers are affected as for `seekoff`. Returns `pos` (the argument) or EOF if the class does not support repositioning or an error occurs. In general a `streampos` should be treated as a “magic cookie” and no arithmetic should be performed on it. Two particular values have special meaning:

```
streampos(0)
```

The beginning of the file.

```
streampos(EOF)
```

Used as an error indication.

### **Buffer flushing**

```
i=sb->sync()
```

Establishes consistency between the internal data structures and the external source or sink. The details of this function depend on the derived class. Usually this “flushes” any characters that have been stored but not yet consumed, and “gives back” any characters that may have been produced but not yet fetched. `sync` returns EOF to indicate errors.

### **Buffer information**

```
i=sb->in_avail()
```

Returns the number of characters that are immediately available in the get area for fetching. `i` characters may be fetched with a guarantee that no errors will be reported.

```
i=sb->out_waiting()
```

Returns the number of characters in the put area that have not been consumed (by the ultimate consumer).

### **Buffer setting function**

```
sb1=sb->setbuf(ptr,len,i)
```

Offers the `len` bytes starting at `ptr` as the reserve area. If `ptr` is null or `len` is zero or less, then an unbuffered state is requested. Whether the offered area is used, or a request for unbuffered state is honored depends on details of the derived class. `setbuf` normally returns `sb`, but if it does not accept the offer or honor the request, it returns 0.

### **Protected interface**

`streambufs` implement the buffer abstraction. However, the `streambuf` class itself contains only basic members for manipulating the characters and normally a class derived from `streambuf` will be used. This page describes the interface needed by programmers who are coding a derived class. Broadly speaking there are two kinds of member functions described here. The non-virtual functions are provided for manipulating a `streambuf` in ways that are appropriate in a derived class. Their descriptions reveal details of the implementation that would be inappropriate in the public interface. The virtual functions permit the derived class to specialize the `streambuf` class in ways appropriate to the specific sources and sinks that it is implementing. The descriptions of the virtual functions explain the obligations of the virtuals of the derived class. If the virtuals behave as specified, the `streambuf` will behave as specified in the public interface. However, if the virtuals do not behave as specified, then the `streambuf` may not behave properly, and an `iostream` (or any other code) that relies on proper behavior of the `streambuf` may not behave properly either.

### **Constructors:**

```
streambuf()
```

Constructs an empty buffer corresponding to an empty sequence.

```
streambuf(b,len)
```

Constructs an empty buffer and then sets up the reserve area to be the `len` bytes starting at `b`.

## **The Get, Put, and Reserved area**

The protected members of `streambuf` present an interface to derived classes organized around three areas (arrays of bytes) managed cooperatively by the base and derived classes. They are the get area, the put area, and the reserve area (or buffer). The get and the put areas are normally disjoint, but they may both overlap the reserve area, whose primary purpose is to be a resource in which space for the put and get areas can be allocated. The get and the put areas are changed as characters are put into and gotten from the buffer, but the reserve area normally remains fixed. The areas are defined by a collection of `char*` values. The buffer abstraction is described in terms of pointers that point between characters, but the `char*` values must point at chars. To establish a correspondence the `char*` values should be thought of as pointing just before the byte they really point at.

### **Functions to examine the pointers**

```
ptr=sb->base()
```

Returns a pointer to the first byte of the reserve area. Space between `sb->base()` and `sb->ebuf()` is the reserve area.

```
ptr=sb->eback()
```

Returns a pointer to a lower bound on `sb->gptr()`. Space between `sb->eback()` and `sb->gptr()` is available for putback.

```
ptr=sb->ebuf()
```

Returns a pointer to the byte after the last byte of the reserve area.

```
ptr=sb->egptr()
```

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

```
ptr=sb->epptr()
```

Returns a pointer to the byte after the last byte of the put area.

```
ptr=sb->egptr()
```

Returns a pointer to the first byte of the get area. The available characters are those between `sb->gptr()` and `sb->egptr()`. The next character fetched will be `*(sb->gptr())` unless `sb->egptr()` is less than or equal to `sb->gptr()`.

```
ptr=sb->pbase()
```

Returns a pointer to the put area base. Characters between `sb->pbase()` and `sb->pptr()` have been stored into the buffer and not yet consumed.

```
ptr=sb->pptr()
```

Returns a pointer to the first byte of the put area. The space between `sb->pptr()` and `sb->epptr()` is the put area and characters will be stored here.

## **Functions for setting the pointers**

Note that to indicate that a particular area (get, put, or reserve) does not exist, all the associated pointers should be set to zero.

```
sb->setb(b, eb, i)
```

Sets base() and ebuf() to b and eb respectively. i controls whether the area will be subject to automatic deletion. If i is non-zero, then b will be deleted when base is changed by another call of setb(), or when the destructor is called for \*sb. If b and eb are both null then we say that there is no reserve area. If b is non-null, there is a reserve area even if eb is less than b and so the reserve area has zero length.

```
sb->setp(p, ep)
```

Sets pptr() to p, pbase() to p, and eptr() to ep.

```
sb->setg(eb, g, eg)
```

Sets eback() to eb, gptr() to g, and egptr() to eg.

## **Other non-virtual members**

```
i=sb->allocate()
```

Tries to set up a reserve area. If a reserve area already exists or if sb->unbuffered() is non-zero, allocate returns 0 without doing anything. If the attempt to allocate space fails, allocate returns EOF, otherwise (allocation succeeds) allocate returns 1. allocate is not called by any non-virtual member function of streambuf.

```
i=sb->blen()
```

Returns the size (in chars) of the current reserve area.

```
dbp()
```

Writes directly on file descriptor 1 information in ASCII about the state of the buffer. It is intended for debugging and nothing is specified about the form of the output. It is considered part of the protected interface because the information it prints can only be understood in relation to that interface, but it is a public function so that it can be called anywhere during debugging.

```
sb->gbump(n)
```

Increments gptr() by n which may be positive or negative. No checks are made on whether the new value of gptr() is in bounds.

```
sb->pbump(n)
```

Increments pptr() by n which may be positive or negative. No checks are made on whether the new value of pptr() is in bounds.

```
sb->unbuffered(i)
i=sb->unbuffered()
```

There is a private variable known as sb's buffering state. sb->unbuffered(i) sets the value of this variable to i and sb->unbuffered() returns the current value. This state is independent of the actual allocation of a reserve area. Its

primary purpose is to control whether a reserve area is allocated automatically by allocate.

## **Virtual member functions**

Virtual functions may be redefined in derived classes to specialize the behavior of streambufs. This section describes the behavior that these virtual functions should have in any derived classes; the next section describes the behavior that these functions are defined to have in base class streambuf.

```
i=sb->doallocate()
```

Is called when allocate determines that space is needed. doallocate is required to call setb to provide a reserve area or to return EOF if it cannot. It is only called if sb->unbuffered() is zero and sb->base() is zero.

```
i=overflow(c)
```

Is called to consume characters. If c is not EOF, overflow also must either save c or consume it. Usually it is called when the put area is full and an attempt is being made to store a new character, but it can be called at other times. The normal action is to consume the characters between pbase() and pptr(), call setp() to establish a new put area, and if c!=EOF store it (using putc). sb->overflow should return EOF to indicate an error; otherwise it should return something else.

```
i=sb->pbackfail(c)
```

Is called when eback() equals gptr() and an attempt has been made to putback c. If this situation can be dealt with (eg by repositioning an external file), pbackfail should return c; otherwise it should return EOF.

```
pos=sb->seekoff(off,dir,mode)
```

Repositions the get and/or put pointers (i.e. the abstract get and put pointers, not pptr() and gptr()). The meanings of off and dir are discussed in the stream class introduction. mode specifies whether the put pointer (ios::out bit set) or the get pointer (ios::in bit set) is to be modified. Both bits may be set in which case both pointers should be affected. A class derived from streambuf is not required to support repositioning. seekoff should return EOF if the class does not support repositioning. If the class does support repositioning, seekoff should return the new position or EOF on error.

```
pos=sb->seekpos(pos,mode)
```

Repositions the streambuf get and/or pointer to pos. mode specifies which pointers are affected as for seekoff. Returns pos (the argument) or EOF if the class does not support repositioning or an error occurs.

```
sb=sb->setbuf(ptr,len)
```

Offers the array at ptr with len bytes to be used as a reserve area. The normal interpretation is that if ptr or len are zero then this is a request to make the sb unbuffered. The derived class may use this area or not as it chooses. It may accept or ignore the request for unbuffered state as it chooses. setbuf should return sb if it honors the request. Otherwise it should return 0.

```
i=sb->sync()
```

Is called to give the derived class a chance to look at the state of the areas, and synchronize them with any external representation. Normally sync should consume any characters that have been stored into the put area, and if possible give back to the source any characters in the get area that have not been fetched. When sync returns there should not be any unconsumed characters, and the get area should be empty. sync should return EOF if some kind of failure occurs.

```
i=sb->underflow()
```

Is called to supply characters for fetching, i.e. to create a condition in which the get area is not empty. If it is called when there are characters in the get area it should return the first character. If the get area is empty, it should create a non-empty get area and return the next character (which it should also leave in the get area). IF there are no more characters available, underflow should return EOF and leave an empty get area.

### **The default definitions of the virtual functions:**

```
i=sb->streambuf::doallocate()
```

Attempts to allocate a reserve area using operator new.

```
i=sb->streambuf::overflow(c)
```

Is compatible with the old stream package, but that behavior is not considered part of the specification of the istream package. Therefore, streambuf::overflow should be treated as if it had undefined behavior. That is, derived classes should always define it.

```
i=sb->streambuf::pbackfail(c)
```

Returns EOF.

```
pos=sb->streambuf::seekpos(pos,mode)
Returns sb->seekoff(streamoff(pos), ios::beg,mode).
```

Thus to define seeking in a derived class, it is frequently only necessary to define seekoff and use the inherited streambuf::seekpos.

```
pos=sb->streambuf::seekoff(off,dir,mode)
```

Returns EOF.

```
sb=sb->streambuf::setbuf(ptr,len)
```

Will honor the request when there is no reserve area.

```
i=sb->streambuf::sync()
```

Returns 0 if the get area is empty and there are no unconsumed characters. Otherwise it returns EOF.

```
i=sb->streambuf::underflow()
```

Is compatible with the old stream package, but that behavior is not considered part of the specification of the istream package. Therefore, streambuf::underflow should be treated as if it had undefined behavior. That is, it should always be defined in derived classes.

**Warning:**

The constructors are public for compatibility with the old stream package. They ought to be protected. The interface for unbuffered actions is awkward. It's hard to write underflow and overflow virtuals that behave properly for unbuffered streambufs without special casing. Also there is no way for the virtuals to react sensibly to multi-character gets or puts.

Although the public interface to streambufs deals in characters and bytes, the interface to derived classes deals in chars. Since a decision had to be made on the types of the real data pointers, it seemed easier to reflect that choice in the types of the protected members than to duplicate all the members with both plain and unsigned char versions. But perhaps all these uses of char\* ought to have been with a typedef.

The implementation contains a variant of setbuf that accepts a third argument. It is present only for compatibility with the old stream package.

## strstreambuf - streambuf specialized to arrays

---

```
streambuf();
strstreambuf(char*, int, char*);
strstreambuf(int);
strstreambuf(unsigned char*,
               int,
               unsigned char*);
strstreambuf(void* (*a)(long),
              void(*f)(void*));
void      freeze(int n=1) ;
char*     str();
streambuf* setbuf(char*, int);
```

A strstreambuf is a streambuf that uses an array of bytes (a string) to hold the sequence of characters. Given the convention that a char\* should be interpreted as pointing just before the char it really points at, the mapping between the abstract get/put pointers (see stream class introduction) and char\* pointers is direct. Moving the pointers corresponds exactly to incrementing and decrementing the char\* values.

To accommodate the need for arbitrary length strings strstreambuf supports a dynamic mode. When a strstreambuf is in dynamic mode, space for the character sequence is allocated as needed. When the sequence is extended too far, it will be copied to a new array.

### **Constructors**

```
strstreambuf()
```

Constructs an empty `strstreambuf` in dynamic mode. This means that space will be automatically allocated to accommodate the characters that are put into the `strstreambuf` (using operators `new` and `delete`). Because this may require copying the original characters, it is recommended that when many characters will be inserted, the program should use `setbuf` (described below) to inform the `strstreambuf`.

```
strstreambuf(a,f)
```

Constructs an empty `strstreambuf` in dynamic mode. `a` is used as the allocator function in dynamic mode. The argument passed to `a` will be a long denoting the number of bytes to be allocated. If `a` is null, operator `new` will be used. `f` is used to free (or delete) areas returned by `a`. The argument to `f` will be a pointer to the array allocated by `a`. If `f` is null, operator `delete` is used.

```
strstreambuf(n)
```

Constructs an empty `strstreambuf` in dynamic mode. The initial allocation of space will be at least `n` bytes.

```
strstreambuf(ptr,n,pstart)
```

Constructs a `strstreambuf` to use the bytes starting at `ptr`. The `strstreambuf` will be in static mode; it will not grow dynamically. If `n` is positive, then the `n` bytes starting at `ptr` are used as the `strstreambuf`. If `n` is zero, `ptr` is assumed to point to the beginning of a null terminated string and the bytes of that string (not including the terminating null character) will constitute the `strstreambuf`. If `n` is negative, the `strstreambuf` is assumed to continue indefinitely. The get pointer is initialized to `ptr`. The put pointer is initialized to `pstart`. If `pstart` is null, then stores will be treated as errors. If `pstart` is non-null, then the initial sequence for fetching (the get area) consists of the bytes between `ptr` and `pstart`. If `pstart` is null, then the initial get area consists of the entire array.

## **Public member functions**

```
ssb->freeze(n)
```

Inhibits (when `n` is non-zero) or permits (when `n` is zero) automatic deletion of the current array. Deletion normally occurs when more space is needed or when `ssb` is being destroyed. Only space obtained via dynamic allocation is ever freed. It is an error (and the effect is undefined) to store characters into a `strstreambuf` that was in dynamic allocation mode and is now frozen. It is possible, however, to thaw (unfreeze) such a `strstreambuf` and resume storing characters.

```
ptr=ssb->str()
```

Returns a pointer to the first char of the current array and freezes `ssb`. If `ssb` was constructed with an explicit array, `ptr` will point to that array. If `ssb` is in dynamic allocation mode, but nothing has yet been stored, `ptr` may be null.

```
ssb->setbuf(0,n)
```

`ssb` remembers `n` and the next time it does a dynamic mode allocation, it makes sure that at least `n` bytes are allocated.

## filebuf - streambuf specialized to files

---

The class filebuf provides a specialized derivation of streambuf for handling files.

## stdiobuf - istream specialized to stdio FILE

---

The class stdiobuf provides a specialized derivation of streambuf for vectoring input and output to C stdio functions.

```
stdiobuf(FILE* f);
FILE*    stdiofile();
```

Operations on a stdiobuf are reflected on the associated FILE. A stdiobuf is constructed in unbuffered mode, which causes all operations to be reflected immediately in the FILE. Calls to seekg and seekp are translated into calls to fseek. setbuf has its usual meaning; if it supplies a reserve area, buffering will be turned back on.

**Warning: stdiobuf is intended to be used when mixing C and C++ code. New C++ code should prefer to use filebufs, which have better performance.**

## strstream - istream specialized to arrays

---

```
enum    open_mode {
        in,
        out,
        ate,
        app,
        trunc,
        nocreate,
        noreplace
} ;

istream(char*) ;
istream(char*, int) ;
strstreambuf*  rdbuf() ;

class ostrstream : public ostream {
public:
    ostrstream();
    ostrstream(char*, int, int=ios::out) ;
    int    pcount() ;
    strstreambuf*  rdbuf() ;
    char*    str();

    class strstream : public strstreambase,
```

```
public istream {

public:
    strstream();
    strstream(char*, int, int mode);
    strstreambuf* rdbuf() ;
    char*    str();
```

strstream specializes istream for “incore” operations, that is, storing and fetching from arrays of bytes. The streambuf associated with a strstream is a strstreambuf (see stream class introduction).

### **Constructors**

```
    istrstream(cp)
```

Characters will be fetched from the (null-terminated) string cp. The terminating null character will not be part of the sequence. Seeks (istream::seekg()) are allowed within that space.

```
    istrstream(cp,len)
```

Characters will be fetched from the array beginning at cp and extending for len bytes. Seeks (istream::seekg()) are allowed anywhere within that array.

```
    ostrstream()
```

Space will be dynamically allocated to hold stored characters.

```
    ostrstream(cp,n,mode)
```

Characters will be stored into the array starting at cp and continuing for n bytes. If ios::ate or ios::app are set in mode, cp is assumed to be a null-terminated string and storing will begin at the null character. Otherwise storing will begin at cp. Seeks are allowed anywhere in the array.

```
    strstream()
```

Space will be dynamically allocated to hold stored characters.

```
    strstream(cp,n,mode)
```

Characters will be stored into the array starting at cp and continuing for n bytes. If ios::ate or ios::app is set in mode, cp is assumed to be a null-terminated string and storing will begin at the null character. Otherwise storing will begin at cp. Seeks are allowed anywhere in the array.

### **istrstream members**

```
    ssb = iss.rdbuf()
```

Returns the strstreambuf associated with iss.

### **ostrstream members**

```
    ssb = oss.rdbuf()
```

Returns the strstreambuf associated with oss.

```
    cp=oss.str()
```

Returns a pointer to the array being used and “freezes” the array. Once `str` has been called the effect of storing more characters into `oss` is undefined. If `oss` was constructed with an explicit array, `cp` is just a pointer to the array. Otherwise, `cp` points to a dynamically allocated area. Until `str` is called, deleting the dynamically allocated area is the responsibility of `oss`. After `str` returns, the array becomes the responsibility of the user program.

```
i=oss.pcount()
```

Returns the number of bytes that have been stored into the buffer. This is mainly of use when binary data has been stored and `oss.str()` does not point to a null terminated string.

### **strstream members**

```
ssb = ss.rdbuf()
```

Returns the `strstreambuf` associated with `ss`.

```
cp=ss.str()
```

Returns a pointer to the array being used and “freezes” the array. Once `str` has been called the effect of storing more characters into `ss` is undefined. If `ss` was constructed with an explicit array, `cp` is just a pointer to the array. Otherwise, `cp` points to a dynamically allocated area. Until `str` is called, deleting the dynamically allocated area is the responsibility of `ss`. After `str` returns, the array becomes the responsibility of the user program.

# INDEX

## Symbols

`_BIG_INLINE` 9  
`_SMALL_INLINE` 9

## A

`abs`  
     class `bcd` 36  
     class `complex` 30  
`access`  
     class `member` 14  
`acos`  
     class `bcd` 36  
     class `complex` 31  
`active`  
     class `task` 43  
`adjustfield`  
     class `ios` 72, 131  
`allocate`  
     class `streambuf` 150  
`app`  
     class `ios` 85  
`arg`  
     class `complex` 30  
`asin`  
     class `complex` 31  
`assignable streams` 82  
`atan`  
     class `complex` 31  
`ate`  
     class `ios` 85  
`attach`  
     class `ifstream` 85  
`awaited`  
     class `event` 46

## B

`bad`  
     class `ios` 65, 134  
`badbit`  
     class `ios` 134

`base`  
     class `streambuf` 149  
`basefield`  
     class `ios` 72, 131  
`bcd.hpp` 34  
`beg`  
     class `ios` 146, 147  
`begin`  
     class `critical_region` 45  
`binary streams` 86  
`bitalloc`  
     class `ios` 133  
`blen`  
     class `streambuf` 150

## C

`change`  
     class `window` 52  
`check_win`  
     class `window` 53  
`cin` 140  
`class .streambuf`  
     public member functions 154  
`class bcd`  
     constants 34  
     constructors 34, 36  
     friend functions 35, 36  
         binary operators 36  
         relational operators 37  
         stream operators 37  
     header file 34  
     member functions 37  
`class complex`  
     constructors 26, 28  
     error handling 26  
     friend functions 27  
     friend operators 27  
     header file 26  
     public member functions 27  
     public operators 26  
`class critical region` 39  
`class critical_region`  
     constructors 42, 45  
     header file 41  
     protected member functions 42, 45  
     public member functions 42, 45  
`class event` 39  
     constructors 42, 45  
     header file 41  
     protected member functions 42, 46  
     public member functions 42, 45

- class filebuf 155
  - constructors 122
  - protected data members 123
  - protected member functions 123
  - public data members 122
  - public member functions 122
- class fstream
  - constructors 124
  - public member functions 124
- class fstreambase
  - constructors 123
  - public member functions 123
- class ifstream
  - constructors 124
  - public member functions 124
- class ios 59
  - buffer control 133
  - constructors 116, 128
  - data member access 129
  - extending state variables 106
  - format control 129
  - operators 116
  - protected data members 117
  - protected enumerated types 117
  - protected member functions 117
  - public data members 116
  - public enumerated types 116
  - public member functions 116
- class iostream
  - constructors 121
- class iostream\_withassign
  - constructors 122
  - operators 122
- class istream
  - constructors 119, 136
  - operators 119
  - public member functions 119
- class istream\_withassign
  - constructors 121
  - operators 122
- class istrstream
  - constructors 125
  - public member functions 156
- class members
  - redefining 16
- class objects 13
- class ofstream
  - constructors 124
  - public member functions 124
- class ostream
  - constructors 120, 141
  - flushing 142
  - operators 120
  - protected member functions 121
  - public member functions 121
- class ostream\_withassign
  - constructors 122
  - operators 122
- class ostrstream
  - constructors 126
  - public member functions 126, 156
- class palettewindow
  - constructors 50, 54
  - framed window 54
  - header file 48
  - public member functions 50, 54
  - unframed window 54
- class semaphore 39
  - constructors 41, 44
  - header file 41
  - public enumerated types 41, 44
  - public member functions 41, 44
- class stdiobuf 155
- class streambuf 59
  - buffer pointers 104
  - buffering 148
  - constructors 118
  - protected constructors 148
  - protected interface 102
  - protected member functions 118, 148
  - public interface 97
  - public member functions 118, 145
  - reserved area 148
- class strstream
  - constructors 126, 156
  - public member functions 126, 157
- class strstreambase
  - constructors 125
- class strstreambuf
  - constructors 125, 153
  - public member functions 125
- class task 39
  - beginning execution of thread 40
  - constructors 43
  - header file 41
  - init 40
  - protected member functions 41, 43
  - public member functions 41, 43
  - stack requirement 40
- class window
  - constructors 48, 51
  - error handler 49, 53
  - framed window 51
  - full screen object 49, 53

- header file 48
- operators 48, 52
- protected data members 49, 53
- protected member functions 49, 53
- public member functions 48, 52
- unframed window 51
- clear
  - class ios 66, 134
  - class semaphore 44
- close
  - class ifstream 84
  - class window 52
- complex 12
- complex.hpp 26
- coniostr.hpp 48
- conj
  - class complex 30
- console I/O streams 55
  - header file 48
- constants
  - \_BcdMaxDecimals 33
- constructors 20
  - class bcd 34, 36
  - class complex 26, 28
  - class critical\_region 42, 45
  - class event 42, 45
  - class filebuf 122
  - class fstream 124
  - class fstreambase 123
  - class ifstream 124
  - class ios 116, 128
  - class istream 121
  - class istream\_withassign 122
  - class istream 119, 136
  - class istream\_withassign 121
  - class istrstream 125
  - class ofstream 124
  - class ostream 120, 141
  - class ostream\_withassign 122
  - class ostrstream 126
  - class palettewindow 50, 54
  - class semaphore 41, 44
  - class streambuf 118
    - protected 148
  - class strstream 126, 156
  - class strstreambase 125
  - class strstreambuf 125, 153
  - class task 40, 43
  - class window 48, 51
- convertcoords
  - class window 53
- cos

- class complex 30
- cosh
- class complex 31
- cur
  - class ios 146, 147

## D

- data member access
  - class ios 129
- dbp
  - class streambuf 150
- dec
  - class ios 131
- Deriving classes
  - task 39
- destroyed
  - class task 40
- device errors 65
- doallocate
  - class streambuf 151, 152

## E

- eback
  - class streambuf 149
- ebuf
  - class streambuf 149
- egptr
  - class streambuf 149
- encapsulation 15, 18, 23
- end
  - class critical\_region 45
  - class ios 146, 147
- eof
  - class eof 134
  - class ios 65, 134
- epptr
  - class streambuf 149
- error handler
  - class window 49, 53
- error handling
  - class complex 26
  - class ios 134
  - streams 65
- errors 14
- exp
  - class complex 31

## F

- fail
  - class ios 66, 134

- failbit
  - class ios 134
- fill
  - class ios 131
- flags
  - class ios 73, 130
- floatfield
  - class ios 72
- floating point precision 77
- flush
  - class ostream 144
- format control 71
  - access functions 129
  - class ios 129
  - conversion base 76, 131
  - defaults 71
  - field width 74
  - flag word 131
  - floating point conversion 132
  - functions 71
  - justification 76, 131
  - manipulators 71
  - padding 131
  - whitespace 71
- format errors 65
- freeze
  - class stringstream 154
- friend functions 20
  - class bcd
    - metric 35
    - transcendental 35
    - type conversion 35
  - class complex
    - arithmetic 27
    - comparison 27
    - decomposition 27, 30
    - hyperbolic 28, 31
    - inverse trigonometric 28, 30
    - metric 27
    - miscellaneous 27, 31
    - miscellaneous transcendental 28
    - polar 27
    - trigonometric 27, 30
- fstream.hpp 114
- full screen object
  - class window 49, 53

## G

- gbump
  - class streambuf 150

- gcount
  - class istream 139
- get
  - class istream 62, 87, 138
- get\_priority
  - class task 43
- get\_sem
  - class critical\_region 45
  - class event 46
- getline
  - class istream 62, 139
- good
  - class ios 67, 134
- goodbit
  - class ios 134

## H

- handle
  - class window 53
- hardfail
  - class ios 134
- hex
  - class ios 131
- hide
  - class window 52

## I

- ignore
  - class istream 139
- in
  - class ios 85
- incore formatting 88
  - storage allocation 88
- info
  - class window 53
- inheritance 18
- init
  - class task 44
- initialization 22
- inline 9
- insertion operator
  - class bcd 37
- instances 12
- io\_state
  - class ios 66
- iomanip.hpp 93, 114
- iostream.hpp 9, 109, 114
- ipfx
  - class istream 136, 137
- istop

- class window 53
- isused
  - class window 53
- iword
  - class ios 107, 133

## K

- kill
  - class task 43

## L

- linking libraries 9
- Lock 39
- log
  - class complex 31

## M

- magnitudes 12
- manipulator
  - dec 145
  - endl 145
  - ends 145
  - flush 145
  - hex 145
  - oct 145
- manipulators
  - class ostream 144
  - dec 140
  - defining 90
  - hex 140
  - IMANIP 95
  - IOANIP 94
  - oct 140
  - OMANIP 93
  - parameterized 91
  - SMANIP 94
  - ws 140
- mathematical functions 24

## N

- naming Conventions 12
- nocreate
  - class ios 85
- noreplace
  - class ios 85
- norm
  - class complex 30
- notify
  - class event 46

- numeric values
  - extraction 76

## O

- obscuredat
  - class window 53
- oct
  - class ios 131
- open
  - class ifstream 84
- open\_mode
  - class ios 85, 135
- operator chaining 61
- operator unary+
  - class complex 29
- operator unary-
  - class complex 29
- operator void \*
  - class ios 67
- operator!=
  - class complex 29
- operator\*
  - class complex 29
- operator\*=
  - class complex 29
- operator+
  - class bcd 37
  - class complex 29
- operator+=
  - class bcd 37
  - class complex 28
- operator-
  - class complex 29
- operator-=
  - class complex 28
- operator/
  - class complex 29
- operator/=
  - class complex 29
- operator<<
  - class bcd 37
- operator=
  - class window 52
- operator==
  - class bcd 37
- operator>>
  - class istream 137
- operators
  - << 61, 142
  - >> 61, 136
  - class bcd

- assignment 34
- friends 34, 35
- relational 37
- stream 37
- unary 34
- class complex 28
  - assignment 26, 28
  - binary 29
  - comparison 29
  - extraction 24
  - insertion 24
  - unary 26, 29
- class ios 116
- class istream\_withassign 122
- class istream 119
- class istream\_withassign 122
- class ostream 120
- class ostream\_withassign 122
- class window 48, 52
- extraction 61, 136
- friends 21
- insertion 61, 142
- members 20
- opfx
  - class ostream 141
- osfx
  - class ostream 141
- out
  - class ios 85
- out\_waiting
  - class streambuf 148
- overflow
  - class streambuf 103, 105, 151, 152
- overloading 24

## P

- palettcolorused
  - class palettewindow 55
- pbackfail
  - class streambuf 151, 152
- pbase
  - class streambuf 149
- pbump
  - class streambuf 150
- pcount
  - class istrstream 157
- peek
  - class istream 63, 139
- polar
  - class complex 30

- pow
  - class complex 31
- pptr
  - class streambuf 149
- precision
  - class ios 132
- predefined streams 61
- proc\_adr
  - class task 40, 44
- protected data members
  - class filebuf 123
  - class ios 117
  - class window 49, 53
- protected enumerated types
  - class ios 117
- protected member functions
  - class critical\_region 42, 45
  - class event 42, 46
  - class filebuf 123
  - class ios 117
  - class istream 120
  - class ostream 121
  - class streambuf 118, 148
  - class task 43
  - class window 49, 53
- public data members
  - class filebuf 122
  - class ios 116
- public enumerated types
  - class ios 116
  - class semaphore 41, 44
- public functions
  - class ios 116
- public member functions
  - class complex 27
  - class critical\_region 42, 45
  - class event 42, 45
  - class filebuf 122
  - class fstream 124
  - class fstreambase 123
  - class ifstream 124
  - class istream 119
  - class istrstream 156
  - class ofstream 124
  - class ostream 121
  - class ostrstream 126, 156
  - class palettewindow 50, 54
  - class semaphore 41, 44
  - class streambuf 118, 145
  - class strstream 126, 157
  - class strstreambuf 125, 154
  - class task 41, 43

- class window 48, 52
- put
  - class ostream 62, 86, 143
- putback
  - class istream 63, 139
- putbeneath
  - class window 52
- putontop
  - class window 52
- pwd
  - class ios 133

## R

- rdbuf
  - class istrstream 156
  - class ostrstream 156
  - class strstream 157
- rdstate
  - class ios 66, 134
- read
  - class istream 62, 87, 139
- real
  - class bcd 36
  - class complex 30
- request
  - class semaphore 45
- reset
  - class event 46
- restrictions 14

## S

- sbumpc
  - class streambuf 145
- seek\_dir
  - class ios 135
  - class streambuf 146
- seekg
  - class fstream 86
  - class istream 140
- seekoff
  - class streambuf 147, 151, 152
- seekp
  - class ostream 144
- seekpos
  - class streambuf 147, 151, 152
- sem\_state
  - class semaphore 44
- set
  - class semaphore 44
- set\_and\_wait

- class semaphore 44
- set\_priority
  - class task 40, 43
- setb
  - class streambuf 150
- setbuf
  - class streambuf 105, 148, 151, 152
  - class strstreambuf 154
- setf
  - class ios 72, 130
- setfill
  - class ios 131
- setframe
  - class window 52
- setg
  - class streambuf 150
- setp
  - class streambuf 150
- setpalette
  - class palettewindow 55
- settitle
  - class window 52
- setw
  - class ios 130
  - class ostream 74
- sgetc
  - class streambuf 146
- sgetn
  - class streambuf 146
- showbase
  - class ios 77
- sin
  - class complex 30
- sinh
  - class complex 31
- skipws
  - class ios 73, 133
- snextc
  - class streambuf 145
- sputbackc
  - class streambuf 146
- sputc
  - class streambuf 146
- sputn
  - class streambuf 146
- sqrt
  - class complex 31
- start
  - class task 40, 43
- state variable 65, 66
- format control 71
- restoring defaults 74

- stdio
  - class ios 142
- stdiostr.hpp 114
- stop
  - class task 40, 43
- stoss
  - class streambuf 146
- str
  - class istrstream 156
  - class strstream 88, 157
  - class strstreambuf 154
- stream class hierarchy 59
- stream flushing 69
- stream positioning 63, 85, 100, 135, 146
- stream.hpp 109
- streamoff
  - class streambuf 147
- streampos 86
  - class streambuf 147
- strings
  - extraction 75
- strstream positioning 144
- strstream.hpp 109, 114
- sync
  - class istream 139
  - class streambuf 106, 147, 152
- sync\_with\_stdio
  - class ios 135

## T

- tan
  - class complex 30
- tanh
  - class complex 31
- task.hpp 41
- task\_id
  - class task 43
- tell
  - class istream 140
- textwin.hpp 48
- tie
  - class istream 69
  - ios 135
- trigger
  - class event 46
- type converters 21
- types
  - arithmetic 19
  - complex 19
  - constructed 23
  - defining new 19

## U

- unbuffered
  - class streambuf 150
- underflow
  - class streambuf 103, 106, 152
- unformatted output 62
- unitbuf
  - class ios 142
- Unlock 39
- unsetf
  - class ios 72
- use
  - class window 52

## W

- wait
  - class event 46
  - class semaphore 44
- whitespace skipping 62, 133
- width
  - class ios 130
  - class ostream 74
- write
  - class ostream 62, 87, 143

## X

- xalloc
  - class ios 106, 133