FORMATTED I/O AND I/O MANIPULATORS

Formatted I/O

The C++ I/O system allows you to format I/O operations. For example, you can set a field width, specify a number base, or determine how many digits after the decimal point will be displayed. There are two related but conceptually different ways that you can format data. First, you can directly access members of the **ios** class. Specifically, you can set various format status flags defined inside the **ios** class or call various **ios** member functions. Second, you can use special functions called *manipulators* that can be included as part of an I/O expression. We will begin the discussion of formatted I/O by using the **ios** member functions and flags.

Formatting Using the ios Members

Each stream has associated with it a set of format flags that control the way information is formatted. The **ios** class declares a bitmask enumeration called **fmtflags** in which the following values are defined. These values are used to set or clear the format flags. If you are using an older compiler, it may not define the **fmtflags** enumeration type. In this case, the format flags will be encoded into a long integer. When the **skipws** flag is set, leading white-space characters (spaces, tabs, and newlines) are discarded when performing input on a stream. When **skipws** is cleared, white-space characters are not discarded.When the **left** flag is set, output is left justified. When **right** is set, output is right justified. When the **internal** flag is set, a numeric value is padded to fill a field by inserting spaces between any sign or base character. If none of these flags are set, output is right justified by default.By default, numeric values are output in

decimal. However, it is possible to change the number base. Setting the **oct** flag causes output to be displayed in octal. Setting the **hex** flag causes output to be displayed in hexadecimal. To return output to decimal, set the **dec** flag.Setting **showbase** causes the base of numeric values to be shown.

For example, if the conversion base is hexadecimal, the value 1F will be displayed as 0x1F. By default, when scientific notation is displayed, the **e** is in lowercase. Also, when a hexadecimal value is displayed, the **x** is in lowercase.

When **uppercase** is set, these characters are displayed in uppercase.

Setting **showpos** causes a leading plus sign to be displayed before positive values.

Setting **showpoint** causes a decimal point and trailing zeros to be displayed for all floating-point output—whether needed or not.

By setting the **scientific** flag, floating-point numeric values are displayed using scientific notation. When **fixed** is set, floating-point values are displayed using normal notation. When neither flag is set, the compiler chooses an appropriate method. When **unitbuf** is set, the buffer is flushed after each insertion operation. When **boolalpha** is set, Booleans can be input or output using the keywords **true** and **false**. Since it is common to refer to the **oct**, **dec**, and **hex** fields, they can be collectively referred to as **basefield**. Similarly, the **left**, **right**, and **internal** fields can be referred to as **adjustfield**. Finally, the **scientific** and **fixed** fields can be referenced as **floatfield**.

I/O manipulators

Each stream has associated with it a set of format flags that control the way information is formatted. The **ios_base** class declares a bitmask enumeration called **fmtflags** in which the following values are defined.

adjustfield basefield boolalpha dec
fixed floatfield hex internal
left oct right scientific
showbase showpoint showpos skipws
unitbuf uppercase

These values are used to set or clear the format flags, using functions such as **setf( )** and **unsetf( )**. In addition to setting or clearing the format flags directly, you may alter the format parameters of a stream through the use of special functions called manipulators, which can be included in an I/O expression.

Several Data Types

In addition to the **fmtflags** type just described, the Standard C++ I/O system defines several other types.

The streamsize and streamoff Types

An object of type **streamsize** is capable of holding the largest number of bytes that will be transferred in any one I/O operation. It is typically some form of integer. An object of type **streamoff** is capable of holding a value that indicates an offset position within a stream. It is typically some form of integer. These types are defined in the header **<ios>**, which is automatically included by the I/O system.

The streampos and wstreampos Types

An object of type **streampos** is capable of holding a value that represents a position within a **char** stream. The **wstreampos** type is capable of holding a value that represents a position with a **wchar_t** stream. These are defined in **<iosfwd>**, which is automatically included by the I/O system.

The pos_type and off_type Types

The types **pos_type** and **off_type** create objects (typically integers) that are capable of holding a value that represents the position and an offset, respectively, within a stream. These types are defined by **ios** (and other classes) and are essentially the same as **streamoff** and **streampos** (or their wide-character equivalents).

The openmode Type

The type **openmode** is defined by **ios_base** and describes how a file will be opened.

The iostate Type

The current status of an I/O stream is described by an object of type **iostate**, which is an enumeration defined by **ios_base** that includes these members.

Name	Meaning
goodbit	No errors occurred.
eofbit	End-of-file is encountered.
Failbit	A nonfatal I/O error has occurred.
Badbit	A fatal I/O error has occurred.

The seekdir Type

The **seekdir** type describes how a random-access file operation will take place. It is defined within **ios_base**. Its valid values are shown here.

beg	Beginning-of-file
cur	Current location
end	End-of-file

The failure Class

In **ios_base** is defined the exception type **failure**. It serves as a base class for the types of exceptions that can be thrown by the I/O system. It inherits **exception** (the standard exception class). The **failure** class has the following constructor: explicit failure(const string &*str*); Here, *str* is a message that describes the error. This message can be obtained from a **failure** object by calling its **what( )** function, shown here: virtual const char *what( ) const throw( );