
15.7. Formatting

Two concepts influence the definition of I/O formats: Most obviously, there are format flags that define, for example, numeric precision, the fill character, or the numeric base. Apart from this, there exists the possibility of adjusting formats to meet special national conventions. This section introduces the format flags. Section 15.8, page 790, and Chapter 16 describe the aspects of internationalized formatting.

15.7.1. Format Flags

The classes `ios_base` and `basic_ios<>` have several members that are used for the definition of various I/O formats. For example, some members store the minimum field width or the precision of floating-point numbers or the fill character. A member of type `ios::fmtflags` stores configuration flags defining, for example, whether positive numbers should be preceded by a positive sign or whether Boolean values should be printed numerically or as words.

Some of the format flags form groups. For example, the flags for octal, decimal, and hexadecimal formats of integer numbers form a group. Special masks are defined to make dealing with such groups easier.

Several member functions can be used to handle all the format definitions of a stream (see Table 15.10). The functions `setf()` and `unsetf()` set or clear, respectively, one or more flags. You can manipulate multiple flags at once by combining them, using the "binary or" operator; that is, operator `|`. The function `setf()` can take a mask as the second argument to clear all flags in a group before setting the flags of the first argument, which are also limited to a group. This does not happen with the version of `setf()` that takes only one argument. For example:

Click here to view code image

```
// set flags showpos and uppercase
std::cout.setf (std::ios::showpos | std::ios::uppercase);

// set only the flag hex in the group basefield
std::cout.setf (std::ios::hex, std::ios::basefield);

// clear the flag uppercase
std::cout.unsetf (std::ios::uppercase);
```

Table 15.10. Member Function to Access Format Flags

Member Function	Meaning
`setf(`*flags*`)`	Sets *flags* as additional flags and returns the previous state of all flags
`setf(`*flags, grp*`)`	Sets *flags* as the new flags of the group identified by *grp* and returns the previous state of all flags
`unsetf(`*flags*`)`	Clears *flags*
`flags()`	Returns all set format flags
`flags(`*flags*`)`	Sets *flags* as the new flags and returns the previous state of all flags
`copyfmt(`*stream*`)`	Copies *all* format definitions from *stream*

Using `flags()`, you can manipulate all format flags at once. Calling `flags()` without an argument returns the current format flags. Calling `flags()` with an argument takes this argument as the new state of all format flags and returns the old state. Thus, `flags()` with an argument clears all flags and sets the flags that were passed. Using `flags()` is useful, for example, for saving the current state of the flags to restore the original state later. The following statements demonstrate an example:

Click here to view code image

```
using std::ios;
using std::cout;

// save current format flags
ios::fmtflags oldFlags = cout.flags();

// do some changes
cout.setf(ios::showpos | ios::showbase | ios::uppercase);
cout.setf(ios::internal, ios::adjustfield);
cout << std::hex << x << std::endl;

// restore saved format flags
cout.flags(oldFlags);
```

By using `copyfmt()` you can copy all the format information from one stream to another. See Section 15.11.1, page 811, for an example.

You can also use manipulators to set and clear format flags. These are presented in Table 15.11.

Table 15.11. Manipulators to Access Format Flags

Manipulator	Effect
`setiosflags(`*flags*`)`	Sets *flags* as format flags (calls `setf(`*flags*`)` for the stream)
`resetiosflags(`*mask*`)`	Clears all flags of the group identified by *mask* (calls `setf(0,`*mask*`)` for the stream)

The manipulators `setiosflags()` and `resetiosflags()` provide the possibility of setting or clearing, respectively, one or more flags in a write or read statement with operator `<<` or `>>`, respectively. To use one of these manipulators, you must include the header file `<iomanip>`. For example:

Click here to view code image

```
#include <iostream>
#include <iomanip>
...
std::cout << resetiosflags(std::ios::adjustfield)  // clear adjustm. flags
          << setiosflags(std::ios::left);           // left-adjust values
```

Some flag manipulations are performed by specialized manipulators. These manipulators are used often because they are more convenient and more readable. They are discussed in the following subsections.

15.7.2. Input/Output Format of Boolean Values

The `boolalpha` flag defines the format used to read or to write Boolean values. It defines whether a numeric or a textual representation is used for Boolean values (Table 15.12).

Table 15.12. Flag for Boolean Representation

Flag	Meaning
`boolalpha`	If set, specifies the use of textual representation; if not set, specifies the use of numeric representation

If the flag is not set (the default), Boolean values are represented using numeric strings. In this case, the value `0` is always used for `false`, and the value `1` is always used for `true`. When reading a Boolean value as a numeric string, it is considered to be an error (setting `failbit` for the stream) if the value differs from `0` or `1`.

If the flag is set, Boolean values are written using a textual representation. When a Boolean value is read, the string has to match the textual representation of either `true` or `false`. The stream's locale object is used to determine the strings used to represent `true` and `false` (see Section 15.8, page 790, and Section 16.2.2, page 865). The standard `"C"` locale object uses the strings `"true"` and `"false"` as representations of the Boolean values.

Special manipulators are defined for the convenient manipulation of this flag (Table 15.13).

Table 15.13. Manipulators for Boolean Representation

Manipulator	Meaning
`boolalpha`	Forces textual representation (sets the flag `ios::boolalpha`)
`noboolalpha`	Forces numeric representation (clears the flag `ios::boolalpha`)

For example, the following statements print `b` first in numeric and then in textual representation:

```
bool b;
...
std::cout << std::noboolalpha << b << " == "
          << std::boolalpha << b << std::endl;
```

15.7.3. Field Width, Fill Character, and Adjustment

Two member functions are used to define the field width and the fill character: `width()` and `fill()` (Table 15.14).

Table 15.14. Member Functions for the Field Width and the Fill Character

Member Function	Meaning
`width()`	Returns the current field width
`width(`*val*`)`	Sets the field width for the next formatted output to *val* and returns the previous field width
`fill()`	Returns the current fill character
`fill(`*c*`)`	Defines *c* as the fill character and returns the previous fill character

Using Field Width, Fill Character, and Adjustment for Output

For the output, `width()` defines a minimum field. This definition applies only to the next formatted field written. Calling `width()` without arguments returns the current field width. Calling `width()` with an integral argument changes the width and returns the former value. The default value for the minimum field width is `0`, which means that the field may have any length. This is also the value to which the field width is set after a value was written.

Note that the field width is never used to truncate output. Thus, you can't specify a maximum field width. Instead, you have to program it. For example, you could write to a string and output only a certain number of characters.

The member function `fill()` defines the fill character that is used to fill the difference between the formatted representation of a value and the minimum field width. The default fill character is a space.

To adjust values within a field, three flags are defined, as shown in Table 15.15. These flags are defined in the class `ios_base` together with the corresponding mask.

Table 15.15. Masks to Adjust Values within a Field

Mask	Flag	Meaning
`adjustfield`	`left`	Left-adjusts the value
	`right`	Right-adjusts the value
	`internal`	Left-adjusts the sign and right-adjusts the value
	None	Right-adjusts the value (the default)

Table 15.16 presents the effect of the functions and the flags used for various values. The underscore is used as the fill character.

Table 15.16. Examples of Adjustments

Adjustment	`width()`	-42	0.12	`"Q"`	`'Q'`
`left`	6	`-42___`	`0.12__`	`Q_____`	`Q_____`
`right`	6	`___-42`	`__0.12`	`_____Q`	`_____Q`
`internal`	6	`-___42`	`__0.12`	`_____Q`	`_____Q`

After any formatted I/O operation is performed, the default field width is restored. The values of the fill character and the adjustment remain unchanged until they are modified explicitly.

Several manipulators are defined to handle the field width, the fill character, and the adjustment (Table 15.17).

Table 15.17. Manipulators for Adjustment

Manipulator	Meaning
`setw(`*val*`)`	Sets the field width of the next input and output to *val* (corresponds to `width()`)
`setfill(`*c*`)`	Defines *c* as the fill character (corresponds to `fill()`)
`left`	Left-adjusts the value
`right`	Right-adjusts the value
`internal`	Left-adjusts the sign and right-adjusts the value

The manipulators `setw()` and `setfill()` use an argument, so you must include the header file `<iomanip>` to use them. For example, the statements

Click here to view code image

```
#include <iostream>
#include <iomanip>
...
std::cout << std::setw(8) << std::setfill('_') << -3.14
          << ' ' << 42 << std::endl;
std::cout << std::setw(8) << "sum: "
          << std::setw(8) << 42 << std::endl;
```

produce this output:

```
___-3.14 42
___sum: ______42
```

Using Field Width for Input

You can also use the field width to define the maximum number of characters read when character sequences of type `char*` are read. If the value of `width()` is not `0`, at most `width()-1` characters are read.

Because ordinary C-strings can't grow while values are read, `width()` or `setw()` should always be used when reading them with operator `>>`. For example:

```
char buffer[81];

// read, at most, 80 characters:
cin >> setw(sizeof(buffer)) >> buffer;
```

This reads at most `80` characters, although `sizeof(buffer)` is `81` because one character is used for the (terminating) null character, which is appended automatically. Note that the following code is a common error:

```
char* s;
cin >> setw(sizeof(s)) >> s;        // RUNTIME ERROR
```

The reason is that `s` is declared only as a pointer without any storage for characters, and `sizeof(s)` is the size of the pointer instead of the size of the storage to which it points. This is a typical example of the problems you encounter if you use C-strings. By using strings, you won't run into these problems:

```
string buffer;
cin >> buffer;                 // OK
```

15.7.4. Positive Sign and Uppercase Letters

Two format flags are defined to influence the general appearance of numeric values: `showpos` and `uppercase` (Table 15.18).

Table 15.18. Flags Affecting Sign and Letters of Numeric Values

Flag	Meaning
`showpos`	Writes a positive sign on positive numbers
`uppercase`	Uses uppercase letters

Using `ios::showpos` dictates that a positive sign for positive numeric values be written. If the flag is not set, only negative values are written with a sign. Using `ios::uppercase` dictates that letters in numeric values be written using uppercase letters. This flag applies to integers using hexadecimal format and to floating-point numbers using scientific notation. By default, letters are written as lowercase, and no positive sign is written. For example, the statements

Click here to view code image

```
std::cout << 12345678.9 << std::endl;

std::cout.setf (std::ios::showpos | std::ios::uppercase);
std::cout << 12345678.9 << std::endl;
```

produce this output:

```
1.23457e+07
+1.23457E+07
```

Both flags can be set or cleared using the manipulators presented in Table 15.19.

Table 15.19. Manipulators for Sign and Letters of Numeric Values

Manipulator	Meaning
`showpos`	Forces writing a positive sign on positive numbers (sets the flag `ios::showpos`)
`noshowpos`	Forces not writing a positive sign (clears the flag `ios::showpos`)
`uppercase`	Forces uppercase letters (sets the flag `ios::uppercase`)
`nouppercase`	Forces lowercase letters (clears the flag `ios::uppercase`)

15.7.5. Numeric Base

A group of three flags defines which base is used for I/O of integer values. The flags are defined in the class `ios_base` with the corresponding mask (Table 15.20).

Table 15.20. Flags Defining the Base of Integral Values

Mask	Flag	Meaning
`basefield`	`oct`	Writes and reads octal
	`dec`	Writes and reads decimal (default)
	`hex`	Writes and reads hexadecimal
	None	Writes decimal and reads according to the leading characters of the integral value

A change in base applies to the processing of all integer numbers until the flags are reset. By default, decimal format is used. There is no support for binary notation. However, you can read and write integral values in binary by using class `bitset` . See Section 12.5.1, page 652, for details.

If none of the base flags is set, output uses a decimal base. If more than one flag is set, decimal is used as the base.

The flags for the numeric base also affect input. If one of the flags for the numeric base is set, all numbers are read using this base. If no flag for the base is set when numbers are read, the base is determined by the leading characters: A number starting with `0x` or `0X` is read as a hexadecimal number. A number starting with `0` is read as an octal number. In all other cases, the number is read as a decimal value.

There are two ways to switch these flags:

1. Clear one flag and set another:

```
std::cout.unsetf (std::ios::dec);
std::cout.setf (std::ios::hex);
```

2. Set one flag and clear all other flags in the group automatically:

Click here to view code image

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

In addition, the C++ standard library provides manipulators that make handling these flags significantly simpler (Table 15.21).

Table 15.21. Manipulators Defining the Base of Integral Values

Manipulator	Meaning
`oct`	Writes and reads octal
`dec`	Writes and reads decimal
`hex`	Writes and reads hexadecimal

For example, the following statements write `x` and `y` in hexadecimal and `z` in decimal:

```
int x, y, z;
...
std::cout << std::hex << x << std::endl;
std::cout << y << ' ' << std::dec << z << std::endl;
```

An additional flag, `showbase` , lets you write numbers according to the usual C/C++ convention for indicating numeric bases of literal values (Table 15.22).

Table 15.22. Flags to Indicate the Numeric Base

Flag	Meaning
`showbase`	If set, indicates the numeric base

If `ios::showbase` is set, octal numbers are preceded by a `0` , and hexadecimal numbers are preceded by `0x` or, if `ios::uppercase` is set, by `0X` . For example, the statements

Click here to view code image

```
std::cout << 127 << ' ' << 255 << std::endl;

std::cout << std::hex << 127 << ' ' << 255 << std::endl;
```

```
std::cout.setf(std::ios::showbase);
std::cout << 127 << ' ' << 255 << std::endl;

std::cout.setf(std::ios::uppercase);
std::cout << 127 << ' ' << 255 << std::endl;
```

produce this output:

```
127 255
7f ff
0x7f 0xff
0X7F 0XFF
```

Note that `ios::showbase` can also be manipulated using the manipulators presented in Table 15.23.

Table 15.23. Manipulators to Indicate the Numeric Base

Manipulator	Meaning
`showbase`	Indicates numeric base (sets the flag `ios::showbase`)
`noshowbase`	Does not indicate numeric base (clears the flag `ios::showbase`)

15.7.6. Floating-Point Notation

Several flags and members control the output of floating-point values. The flags, presented in Table 15.24, define whether output is written using decimal or scientific notation. These flags are defined in the class `ios_base` together with the corresponding mask. If `ios::fixed` is set, floating-point values are printed using decimal notation. If `ios::scientific` is set, scientific — that is, exponential — notation is used.

Table 15.24. Flags for Floating-Point Notation

Mask	Flag(s)	Meaning
`floatfield`	`fixed`	Uses decimal notation
	`scientific`	Uses scientific notation
	None	Uses the “best” of these two notations (default)
	`fixed|scientific`	Hexadecimal scientific notation (since C++11)

Before C++11, specifying `fixed|scientific` was not defined. Since C++11, this can be used to define a hexadecimal scientific notation, which also the format specifier `%a` provides for `printf()` : a hexadecimal value to the power of `2.` For example, `234.5` is written as `0x1.d5p+7` (`0x1.d5` times 2^7, which is $1*\frac{128}{1} + 13*\frac{128}{16} + 5*\frac{128}{256}$).

Using the flag `showpoint` , you can force the stream to write a decimal point and trailing zeros until places according to the current precision are written (Table 15.25).

Table 15.25. Flag to Force Decimal Point

Flag	Meaning
`showpoint`	Always writes a decimal point and fills up with trailing zeros

To define the precision, the member function `precision()` is provided (see Table 15.26).

Table 15.26. Member Function for the Precision of Floating-Point Values

Member Function	Meaning
`precision()`	Returns the current precision of floating-point values
`precision(`*val*`)`	Sets *val* as the new precision of floating-point values and returns the old

If scientific notation is used, `precision()` defines the number of decimal places in the fractional part. In all cases, the remainder is not cut off but rounded. Calling `precision()` without arguments returns the current precision. Calling it with an argument sets the precision to that value and returns the previous precision. The default precision is six decimal places.

By default, neither `ios::fixed` nor `ios::scientific` is set. In this case, the notation used depends on the value written. All meaningful but, at most, `precision()` decimal places are written as follows: A leading zero before the decimal point and/or all trailing zeros and potentially even the decimal point are removed. If `precision()` places are sufficient, decimal notation is used; otherwise, scientific notation is used.

Table 15.27 shows the somewhat complicated dependencies between flags and precision, using two concrete values as an example.

Table 15.27. Example of Floating-Point Formatting

	precision()	421.0	0.0123456789
Normal	2	4.2e+02	0.012
	6	421	0.0123457
With showpoint	2	4.2e+02	0.012
	6	421.000	0.0123457
fixed	2	421.00	0.01
	6	421.000000	0.012346
scientific	2	4.21e+02	1.23e-02
	6	4.210000e+02	1.234568e-02
fixed|scientific	2	0x1.a5p+8	0x1.95p-7
	6	0x1.a50000p+8	0x1.948b10p-7

As for integral values, `ios::showpos` can be used to write a positive sign, and `ios::uppercase` can be used to dictate whether the scientific notations should use uppercase or lowercase letters.

The flag `ios::showpoint`, the notation, and the precision can be configured using the manipulators presented in Table 15.28.

Table 15.28. Manipulators for Floating-Point Values

Manipulator	Meaning
showpoint	Always writes a decimal point (sets the flag ios::showpoint)
noshowpoint	Does not require a decimal point (clears the flag showpoint)
setprecision(*val*)	Sets *val* as the new value for the precision
fixed	Uses decimal notation
scientific	Uses scientific notation
hexfloat	Uses hexadecimal scientific notation (since C++11)
defaultfloat	Uses normal notation (clears the flag floatfield, since C++11)

For example, the statement

```
std::cout << std::scientific << std::showpoint
          << std::setprecision(8)
          << 0.123456789 << std::endl;
```

produces this output:

```
1.23456789e-01
```

Note that `setprecision()` is a manipulator with an argument, so you must include the header file `<iomanip>` to use it.

15.7.7. General Formatting Definitions

Two more format flags complete the list of formatting flags: `skipws` and `unitbuf` (Table 15.29).

Table 15.29. Other Formatting Flags

Flag	Meaning
skipws	Skips leading whitespaces automatically when reading a value with operator >>
unitbuf	Flushes the output buffer after each write operation

By default, `ios::skipws` is set, which means that leading whitespaces are skipped by operator `>>`. Often, it is useful to have this flag set. For example, with it set, reading the separating spaces between numbers explicitly is not necessary. However, this implies that reading space characters using operator `>>` is not possible because leading whitespaces are always skipped.

With `ios::unitbuf`, the buffering of the output is controlled. When it is set, output is unbuffered, which means that the output buffer is flushed after each write operation. By default, this flag is not set for most streams. However, for the streams `cerr` and `wcerr` , this flag is set initially.

Both flags can be manipulated using the manipulators presented in Table 15.30.

Table 15.30. Manipulators for Other Formatting Flags

Manipulator	Meaning
`skipws`	Skips leading whitespaces with operator >> (sets the flag `ios::skipws`)
`noskipws`	Does not skip leading whitespaces with operator >> (clears the flag `ios::skipws`)
`unitbuf`	Flushes the output buffer after each write operation (sets the flag `ios::unitbuf`)
`nounitbuf`	Does not flush the output buffer after each write operation (clears the flag `ios::unitbuf`)