
15.6. Manipulators

Manipulators for streams, introduced in Section 15.1.5, page 746, are objects that modify a stream when applied with the standard I/O operators. This does not necessarily mean that something is read or written. The basic manipulators defined in `<istream>` or `<ostream>` are presented in Table 15.8.

Table 15.8. Manipulators Defined in `<istream>` or `<ostream>`

Manipulator	Class	Meaning
`endl`	`basic_ostream`	Inserts a newline character into the buffer and flushes the output buffer to its device
`ends`	`basic_ostream`	Inserts a (terminating) null character into the buffer
`flush`	`basic_ostream`	Flushes the output buffer to its device
`ws`	`basic_istream`	Reads and ignores whitespaces

Manipulators with Arguments

Some of the manipulators process arguments. For example, you can use the following to set the minimum field width of the next output and the fill character:

```
std::cout << std::setw(6) << std::setfill('_');
```

The standard manipulators with arguments are defined in the header file `<iomanip>`, which must be included to work with the standard manipulators taking arguments:

```
#include <iomanip>
```

The standard manipulators taking arguments are all concerned with details of formatting, so they are described when general formatting options (see Section 15.7, page 779), time formatting (see Section 16.4.3, page 890), or monetary formatting (see Section 16.4.2, page 882) are introduced.

15.6.1. Overview of All Manipulators

Table 15.9 gives an overview of all manipulators provided by the C++ standard library, including the page where you can find details. `hexfloat`, `defaultfloat`, `put_time()`, `get_time()`, `put_money()`, and `get_money()` are provided since C++11.

Table 15.9. Manipulators Provided by the C++ Standard Library

Manipulator	Effect	Page
`endl`	Writes a newline character and flushes the output	776
`ends`	Writes a (terminating) null character	774
`flush`	Flushes the output	774
`ws`	Reads and ignores whitespaces	774
`skipws`	Skips leading whitespaces with operator >>	789
`noskipws`	Does not skip leading whitespaces with operator >>	789
`unitbuf`	Flushes the output buffer after each write operation	789
`nounitbuf`	Does not flush the output buffer after each write operation	789
`setiosflags(`*flags*`)`	Sets *flags* as format flags	780
`resetiosflags(`*m*`)`	Clears all flags of the group identified by mask *m*	780
`setw(`*val*`)`	Sets the field width of the next input and output to *val*	783
`setfill(`*c*`)`	Defines *c* as the fill character	783
`left`	Left-adjusts values	783
`right`	Right-adjusts values	783
`internal`	Left-adjusts signs and right-adjusts values	783
`boolalpha`	Forces textual representation for Boolean values	781
`noboolalpha`	Forces numeric representation for Boolean values	781
`showpos`	Forces writing a positive sign on positive numbers	784
`noshowpos`	Forces not writing a positive sign on positive numbers	784
`uppercase`	Forces uppercase letters for numeric values	784
`nouppercase`	Forces lowercase letters for numeric values	784
`oct`	Reads and writes integral values octal	785
`dec`	Reads and writes integral values decimal	785
`hex`	Reads and writes integral values hexadecimal	785
`showbase`	Indicates numeric base of numeric values	786
`noshowbase`	Does not indicate numeric base of numeric values	786
`showpoint`	Always writes a decimal point for floating-point values	788
`noshowpoint`	Does not require a decimal point for floating-point values	788
`setprecision(`*val*`)`	Sets *val* as the new value for the precision of floating-point values	788
`fixed`	Uses decimal notation for floating-point values	788
`scientific`	Uses scientific notation for floating-point values	788
`hexfloat`	Uses hexadecimal scientific notation for floating-point values	788
`defaultfloat`	Uses normal floating-point notation	788
`put_time(`*val*`,`*fmt*`)`	Writes a date/time value according to the format *fmt*	890
`get_time(`*val*`,`*fmt*`)`	Reads a time/date value according to the format *fmt*	890
`put_money(`*val*`)`	Writes a monetary value using the local currency symbol	882
`put_money(`*val*`,`*intl*`)`	Writes a monetary value using the currency symbol according to *intl*	882
`get_money(`*val*`)`	Reads a monetary value using the local currency symbol	882
`get_money(`*val*`,`*intl*`)`	Reads a monetary value using the currency symbol according to *intl*	882

15.6.2. How Manipulators Work

Manipulators are implemented using a very simple trick that not only enables the convenient manipulation of streams but also demonstrates the power provided by function overloading. Manipulators are nothing more than functions passed to the I/O operators as arguments. The functions are then called by the operator. For example, the output operator for class `ostream` is basically overloaded like this:

Click here to view code image

```
ostream & ostream ::operator << ( ostream & (*op)( ostream& ))
{
    // call the function passed as parameter with this stream as the argument
    return (*op)(*this);
```

```
}
```

The argument `op` is a pointer to a function that takes *ostream* as an argument and returns *ostream* (it is assumed that the *ostream* given as the argument is returned). If the second operand of operator `<<` is such a function, this function is called with the first operand of operator `<<` as the argument.

This may sound very complicated, but it is relatively simple. An example should make it clearer. The manipulator — that is, the function — `endl()` for `ostream` is implemented basically like this:

```
std::ostream& std::endl (std::ostream& strm)
{
    // write newline
    strm.put('\n');

    // flush the output buffer
    strm.flush();

    // return strm to allow chaining
    return strm;
}
```

You can use this manipulator in an expression such as the following:

```
std::cout << std::endl
```

Here, operator `<<` is called for stream `cout` with the `endl()` function as the second operand. The implementation of operator `<<` transforms this call into a call of the passed function with the stream as the argument:

```
std::endl(std::cout)
```

The same effect as "writing" the manipulator can also be achieved by calling this expression directly. An advantage to using the function notation is that it is not necessary to provide the namespace for the manipulator:

```
endl(std::cout)
```

The reason is that, according to *ADL* (*argument-dependent lookup*, also known as *Koenig lookup*), functions are looked up in the namespaces where their arguments are defined if they are not found otherwise.

Because the stream classes are class templates parametrized with the character type, the real implementation of `endl()` looks like this:

```
template <typename charT, typename traits>
std::basic_ostream<charT,traits>&
std::endl (std::basic_ostream<charT,traits>& strm)
{
    strm.put(strm.widen('\n'));
    strm.flush();
    return strm;
}
```

The member function `widen()` is used to convert the newline character into the character set currently used by the stream. See Section 15.8, page 790, for more details.

How the manipulators with arguments work exactly is implementation dependent, and there is no standard way to implement user-defined manipulators with arguments (see the next section for an example).

15.6.3. User-Defined Manipulators

To define your own manipulator, you simply need to write a function such as `endl()`. For example, the following function defines a manipulator that ignores all characters until end-of-line:

Click here to view code image

```
// io/ignore1.hpp

#include <istream>
#include <limits>

template <typename charT, typename traits>
inline
std::basic_istream<charT,traits>&
ignoreLine (std::basic_istream<charT,traits>& strm)
{
    // skip until end-of-line
    strm.ignore(std::numeric_limits<std::streamsize>::max(),
```

```
                    strm.widen('\n'));

    // return stream for concatenation
    return strm;
}
```

The manipulator simply delegates the work to the function `ignore()`, which in this case discards all characters until end-of-line (`ignore()` was introduced in Section 15.5.1, page 770).

The application of the manipulator is very simple:

```
// ignore the rest of the line
std::cin >> ignoreLine;
```

Applying this manipulator multiple times enables you to ignore multiple lines:

```
// ignore two lines
std::cin >> ignoreLine >> ignoreLine;
```

This works because a call to the function `ignore(max,c)` ignores all characters until the `c` is found in the input stream, or `max` characters are read or the end of the stream was reached. However, this character is discarded, too, before the function returns.

As written, there are multiple ways to define your own manipulator taking arguments. For example, the following code ignores `n` lines:

Click here to view code image

```
// io/ignore2.hpp

#include <istream>
#include <limits>

class ignoreLine
{
  private:
    int num;
  public:
    explicit ignoreLine (int n=1) : num(n) {
    }

    template <typename charT, typename traits>
    friend std::basic_istream<charT,traits>&
    operator>> (std::basic_istream<charT,traits>& strm,
                const ignoreLine& ign)
    {
        // skip until end-of-line num times
        for (int i=0; i<ign.num; ++i) {
            strm.ignore(std::numeric_limits<std::streamsize>::max(),
                        strm.widen('\n'));
        }

        // return stream for concatenation
        return strm;
    }
};
```

Here, the manipulator `ignoreLine` is a class, which takes the argument to get initialized, and the input operator is overloaded for objects of this class.