15.3. Standard Stream Operators << and >>

In C and C++, operators `<<` and `>>` are used for shifting bits of an integer to the right or the left, respectively. The classes `basic_istream<>` and `basic_ostream<>` overload operators `>>` and `<<` as the standard I/O operators. Thus, in C++, the "shift operators" became the "I/O operators."[7]

[7] Some people also call the I/O operators *inserters* and *extractors*.

15.3.1. Output Operator <<

The class `basic_ostream` — and thus also the classes `ostream` and `wostream` — defines `<<` as an output operator and overloads it for almost all fundamental types, excluding `void` and `nullptr_t`, as well as for `char*` and `void*`.

The output operators for streams are defined to send their second argument to the corresponding stream. Thus, the data is sent in the direction of the arrow:

```
int i = 7;
std::cout << i;             // outputs: 7

float f = 4.5;
std::cout << f;             // outputs: 4.5
```

Operator `<<` can be overloaded such that the second argument is an arbitrary data type, thereby allowing the integration of your own data types into the I/O system. The compiler ensures that the correct function for outputting the second argument is called. Of course, this function should in fact transform the second argument into a sequence of characters sent to the stream.

The C++ standard library also uses this mechanism to provide output operators for specific types, such as strings (see Section 13.3.10, page 712), bitsets (see Section 12.5.1, page 652), and complex numbers (see Section 17.2.3, page 933):

```
std::string s("hello");
s += ", world";
std::cout << s;              // outputs: hello, world

std::bitset<10> flags(7);
std::cout << flags;          // outputs: 0000000111

std::complex<float> c(3.1,7.4);
std::cout << c;              // outputs: (3.1,7.4)
```

The details about writing output operators for your own data types are explained in Section 15.11, page 810.

The fact that the output mechanism can be extended to incorporate your own data types is a significant improvement over C's I/O mechanism, which uses `printf()`: It is not necessary to specify the type of an object to be printed. Instead, the overloading of different types ensures that the correct function for printing is deduced automatically. The mechanism is not limited to standard types. Thus, the user has only one mechanism, and it works for all types.

Operator `<<` can also be used to print multiple objects in one statement. By convention, the output operators return their first argument. Thus, the result of an output operator is the output stream. This allows you to chain calls to output operators as follows:

Click here to view code image

```
std::cout << x << " times " << y << " is " << x * y << std::endl;
```

Operator `<<` is evaluated from left to right. Thus,

```
std::cout << x
```

is executed first. Note that the evaluative order of the operator does not imply any specific order in which the arguments are evaluated; only the order in which the operators are executed is defined. This expression returns its first operand, `std::cout`. So,

```
std::cout << " times "
```

is executed next. The object `y`, the string literal `" is "`, and the result of `x * y` are printed accordingly. Note that the multiplication operator has a higher priority than operator `<<`, so you need no parentheses around `x * y`. However, there are operators that have lower priority, such as all logical operators. In this example, if `x` and `y` are floating-point numbers with the values

`2.4` and `5.1` , the following is printed:

```
2.4 times 5.1 is 12.24
```

Note that since C++11, concurrent output using the same stream object is possible but might result in interleaved characters (see Section 15.2.2, page 752).

15.3.2. Input Operator >>

The class `basic_istream` — and thus also the classes `istream` and `wistream` — defines `>>` as an input operator. Similar to `basic_ostream` , this operator is overloaded for almost all fundamental types, excluding `void` and `nullptr_t` , as well as for `char*` and `void*` . The input operators for streams are defined to store the value read in their second argument. As with operator `<<` , the data is sent in the direction of the arrow:

Click here to view code image

```
int i;
std::cin >> i;     // reads an int from standard input and stores it in i

float f;
std::cin >> f;     // reads a float from standard input and stores it in f
```

Note that the second argument is modified. To make this possible, the second argument is passed by nonconstant reference.

As with output operator `<<` , it is also possible to overload the input operator for arbitrary data types and to chain the calls:

```
float f;
std::complex<double> c;

std::cin >> f >> c;
```

To make this possible, leading whitespace is skipped by default. However, this automatic skipping of whitespace can be turned off (see Section 15.7.7, page 789).

Note that since C++11, concurrent input using the same stream object is possible but might result in characters where it is not defined which thread reads which character (see Section 15.2.2, page 752).

15.3.3. Input/Output of Special Types

The standard I/O operators are provided for almost all fundamental types (excluding `void` and `nullptr_t`) as well as for `char*` , and `void*` . However, special rules apply to some of these types and to user-defined types.

Numeric Types

When reading numeric values, the input must start with at least one digit. Otherwise, the numeric value will be set to `0` and the `failbit` (see Section 15.4.1, page 758) is set:

```
int x;
std::cin >> x;     // assigns 0 to x, if the next character does not fit
```

However, if there is no input or if the `failbit` is set already, calling the input operator will not modify `x` . This also applies to `bool` .

Type `bool`

By default, Boolean values are printed and read numerically: `false` is converted into and from `0` , and `true` is converted into and from `1` . When reading, values different from `0` and `1` are considered to be an error. In this case, the `ios::failbit` is set, which might throw a corresponding exception (see Section 15.4.4, page 762).

It is also possible to set up the formatting options of the stream to use character strings for the I/O of Boolean values (see Section 15.7.2, page 781). This touches on the topic of internationalization: Unless a special locale object is used, the strings `"true"` and `"false"` are used. In other locale objects, different strings might be used. For example, a German locale object would use the strings `"wahr"` and `"falsch"` . See Chapter 16, especially Section 16.2.2, page 865, for more details.

Types `char` and `wchar_t`

When a `char` or a `wchar_t` is being read with operator `>>` , leading whitespace is skipped by default. To read any character, including whitespace, you can either clear the flag `skipws` (see Section 15.7.7, page 789) or use the member function `get()` (see Section 15.5.1, page 768).

Type `char*`

A C-string (that is, a `char*`) is read wordwise. That is, when a C-string is being read, leading whitespace is skipped by default, and the string is read until another whitespace character or end-of-file is encountered. Whether leading whitespace is skipped automatically can be controlled with the flag `skipws` (see Section 15.7.7, page 789).

Note that this behavior means that the string you read can become arbitrarily long. It is already a common error in C programs to assume that a string can be a maximum of 80 characters long. There is no such restriction. Thus, you must arrange for a premature termination of the input when the string is too long. To do this, you should *always* set the maximum length of the string to be read. This normally looks something like the following:

```
char buffer[81];       // 80 characters and '\0'
std::cin >> std::setw(81) >> buffer;
```

The manipulator `setw()` and the corresponding stream parameter are described in detail in Section 15.7.3, page 781.

The type `string` from the C++ standard library (see Chapter 13) grows as needed to accommodate a lengthy string. Rather than using `char*`, this is much easier and less error prone. In addition, `string`s provide a convenient function `getline()` for line-by-line reading (see Section 13.2.10, page 677). So, whenever you can, avoid the use of C-strings and use strings.

Type `void*`

Operators `<<` and `>>` also provide the possibility of printing a pointer and reading it back in again. An address is printed in an implementation-dependent format if a parameter of type `void*` is passed to the output operator. For example, the following statement prints the contents of a C-string and its address:

Click here to view code image

```
char* cstring = "hello";

std::cout << "string \"" << cstring << "\" is located at address: "
          << static_cast<void*>(cstring) << std::endl;
```

The result of this statement might appear as follows:

```
string "hello" is located at address: 0x10000018
```

It is even possible to read an address again with the input operator. However, note that addresses are normally transient. The same object can get a different address in a newly started program. A possible application of printing and reading addresses may be programs that exchange addresses for object identification or programs that share memory.

Stream Buffers

You can use operators `>>` and `<<` to read directly into a stream buffer and to write directly out of a stream buffer respectively. This is probably the fastest way to copy files by using C++ I/O streams. See Section 15.14.3, page 846, for examples.

User-Defined Types

In principle, it is very easy to extend this technique to your own types. However, paying attention to all possible formatting data and error conditions takes more effort than you might think. See Section 15.11, page 810, for a detailed discussion about extending the standard I/O mechanism for your own types.

Monetary and Time Values

Since C++11, it is possible to use manipulators to directly read or write monetary or time values. For example, the following program allows you to write the current date and time, reads a new time, and prints the difference in minutes:

Click here to view code image

```
// io/timemanipulator1.cpp

#include <iostream>
#include <iomanip>
#include <chrono>
#include <ctime>
#include <cstdlib>
using namespace std;

int main ()
{
    // process and print current date and time:
    auto now = chrono::system_clock::now();
    time_t t = chrono::system_clock::to_time_t(now);
    tm* nowTM = localtime(&t);
    cout << put_time(nowTM,"date: %x\ntime: %X\n") << endl;

    // read new time (same date)
    tm time(*nowTM);              // copy date for new time
    cout << "new time [HH:MM]: ";
```

```
        cin >> get_time(&time,"%H:%M");   // read new time
        if (!cin) {
            cerr << "invalid format read" << endl;
            exit(EXIT_FAILURE);
        }
        // process difference in minutes:
        auto tp = chrono::system_clock::from_time_t(mktime(&time));
        auto diff = chrono::duration_cast<chrono::minutes>(tp-now);
        cout << "difference: " << diff.count() << " minutes" << endl;
    }
```

Before asking for a new time, the program might output:

```
date: 09/14/11
time: 11:08:52
```

The corresponding manipulators allow you to take international behavior into account. See Section 16.4.2, page 882, for details of monetary manipulators and Section 16.4.3, page 890, for details of time manipulators. For details of the chrono library (defining `std::chrono::system_time`, `timepoints`, and `durations`), see Section 5.7, page 143.