8

I/O Streams

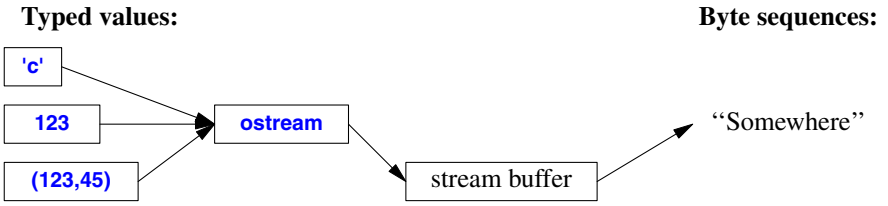
What you see is all you get.
– Brian W. Kernighan

- Introduction
- Output
- Input
- I/O State
- I/O of User-Defined Types
- Formatting
- File Streams
- String Streams
- Advice

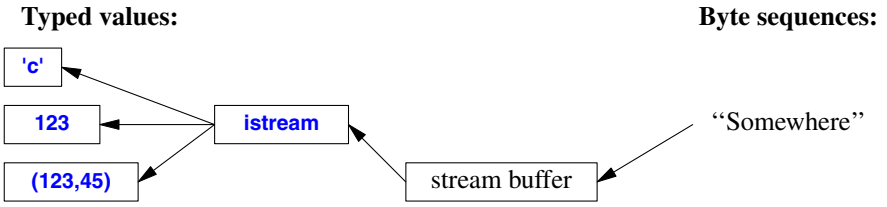
8.1 Introduction

The I/O stream library provides formatted and unformatted buffered I/O of text and numeric values.

An **ostream** converts typed objects to a stream of characters (bytes):



An **istream** converts a stream of characters (bytes) to typed objects:



The operations on **istream**s and **ostream**s are described in §8.3 and §8.2. The operations are type-safe, type-sensitive, and extensible to handle user-defined types.

Other forms of user interaction, such as graphical I/O, are handled through libraries that are not part of the ISO standard and therefore not described here.

These streams can be used for binary I/O, be used for a variety of character types, be locale specific, and use advanced buffering strategies, but these topics are beyond the scope of this book.

8.2 Output

In **<ostream>**, the I/O stream library defines output for every built-in type. Further, it is easy to define output of a user-defined type (§8.5). The operator **<<** ("put to") is used as an output operator on objects of type **ostream**; **cout** is the standard output stream and **cerr** is the standard stream for reporting errors. By default, values written to **cout** are converted to a sequence of characters. For example, to output the decimal number **10**, we can write:

```
void f()
{
    cout << 10;
}
```

This places the character **1** followed by the character **0** on the standard output stream.

Equivalently, we could write:

```
void g()
{
    int i {10};
    cout << i;
}
```

Output of different types can be combined in the obvious way:

```
void h(int i)
{
    cout << "the value of i is ";
    cout << i;
    cout << '\n';
}
```

For **h(10)**, the output will be:

```
the value of i is 10
```

People soon tire of repeating the name of the output stream when outputting several related items. Fortunately, the result of an output expression can itself be used for further output. For example:

```
void h2(int i)
{
    cout << "the value of i is " << i << '\n';
}
```

This `h2()` produces the same output as `h()`.

A character constant is a character enclosed in single quotes. Note that a character is output as a character rather than as a numerical value. For example:

```
void k()
{
    int b = 'b';    // note: char implicitly converted to int
    char c = 'c';
    cout << 'a' << b << c;
}
```

The integer value of the character `'b'` is `98` (in the ASCII encoding used on the C++ implementation that I used), so this will output `a98c`.

8.3 Input

In `<istream>`, the standard library offers `istream`s for input. Like `ostream`s, `istream`s deal with character string representations of built-in types and can easily be extended to cope with user-defined types.

The operator `>>` (“get from”) is used as an input operator; `cin` is the standard input stream. The type of the right-hand operand of `>>` determines what input is accepted and what is the target of the input operation. For example:

```
void f()
{
    int i;
    cin >> i;       // read an integer into i

    double d;
    cin >> d;       // read a double-precision floating-point number into d
}
```

This reads a number, such as `1234`, from the standard input into the integer variable `i` and a floating-point number, such as `12.34e5`, into the double-precision floating-point variable `d`.

Like output operations, input operations can be chained, so I could equivalently have written:

```
void f()
{
        int i;
        double d;
        cin >> i >> d;          // read into i and d
}
```

In both cases, the read of the integer is terminated by any character that is not a digit. By default, `>>` skips initial whitespace, so a suitable complete input sequence would be

```
1234
12.34e5
```

Often, we want to read a sequence of characters. A convenient way of doing that is to read into a `string`. For example:

```
void hello()
{
        cout << "Please enter your name\n";
        string str;
        cin >> str;
        cout << "Hello, " << str << "!\n";
}
```

If you type in `Eric` the response is:

```
Hello, Eric!
```

By default, a whitespace character, such as a space or a newline, terminates the read, so if you enter `Eric Bloodaxe` pretending to be the ill-fated king of York, the response is still:

```
Hello, Eric!
```

You can read a whole line (including the terminating newline character) using the `getline()` function. For example:

```
void hello_line()
{
        cout << "Please enter your name\n";
        string str;
        getline(cin,str);
        cout << "Hello, " << str << "!\n";
}
```

With this program, the input `Eric Bloodaxe` yields the desired output:

```
Hello, Eric Bloodaxe!
```

The newline that terminated the line is discarded, so `cin` is ready for the next input line.

The standard strings have the nice property of expanding to hold what you put in them; you don't have to precalculate a maximum size. So, if you enter a couple of megabytes of semicolons, the program will echo pages of semicolons back at you.

8.4 I/O State

An **iostream** has a state that we can examine to determine whether an operation succeeded. The most common use is to read a sequence of values:

```
vector<int> read_ints(istream& is)
{
    vector<int> res;
    int i;
    while (is>>i)
        res.push_back(i);
    return res;
}
```

This reads from **is** until something that is not an integer is encountered. That something will typically be the end of input. What is happening here is that the operation **is>>i** returns a reference to **is**, and testing an **iostream** yields **true** if the stream is ready for another operation.

In general, the I/O state holds all the information needed to read or write, such as formatting information (§8.6), error state (e.g., has end-of-input been reached?), and what kind of buffering is used. In particular, a user can set the state to reflect that an error has occurred (§8.5) and clear the state if an error wasn't serious. For example, we could imagine reading a sequence of integers than might contain some form of nesting:

```
while (cin) {
    for (int i; cin>>i; ) {
        // ... use the integer ...
    }

    if (cin.eof()) {
        // .. all is well we reached the end-of-file ...
    }
    else if (cin.fail()) {          // a potentially recoverable error
        cin.clear();                // reset the state to good()
        char ch;
        if (cin>>ch) {              // look for nesting represented by { ... }
            switch (ch) {
            case '{':
                // ... start nested structure ...
                break;
            case '}':
                // ... end nested structure ...
                break;
            default:
                cin.setstate(ios_base::failbit);        // add fail() to cin's state
            }
        }
    }
    // ...
}
```

8.5 I/O of User-Defined Types

In addition to the I/O of built-in types and standard **string**s, the **iostream** library allows programmers to define I/O for their own types. For example, consider a simple type **Entry** that we might use to represent entries in a telephone book:

```
struct Entry {
        string name;
        int number;
};
```

We can define a simple output operator to write an **Entry** using a *{"name",number}* format similar to the one we use for initialization in code:

```
ostream& operator<<(ostream& os, const Entry& e)
{
        return os << "{\"" << e.name << "\", " << e.number << "}";
}
```

A user-defined output operator takes its output stream (by reference) as its first argument and returns it as its result.

The corresponding input operator is more complicated because it has to check for correct formatting and deal with errors:

```
istream& operator>>(istream& is, Entry& e)
        // read { "name" , number } pair. Note: formatted with { " " , and }
{
        char c, c2;
        if (is>>c && c=='{' && is>>c2 && c2=='"') { // start with a { "
                string name;                    // the default value of a string is the empty string: ""
                while (is.get(c) && c!='"')     // anything before a " is part of the name
                        name+=c;

                if (is>>c && c==',') {
                        int number = 0;
                        if (is>>number>>c && c=='}') { // read the number and a }
                                e = {name,number};      // assign to the entry
                                return is;
                        }
                }
        }
        is.state_base::failbit);            // register the failure in the stream
        return is;
}
```

An input operation returns a reference to its **istream** which can be used to test if the operation succeeded. For example, when used as a condition, **is>>c** means "Did we succeed at reading from **is** into **c**?"

The **is>>c** skips whitespace by default, but **is.get(c)** does not, so that this **Entry**-input operator ignores (skips) whitespace outside the name string, but not within it. For example:

```
{ "John Marwood Cleese" , 123456        }
{"Michael Edward Palin",987654}
```

We can read such a pair of values from input into an **Entry** like this:

```
for (Entry ee; cin>>ee; )  // read from cin into ee
    cout << ee << '\n';    // write ee to cout
```

The output is:

```
{"John Marwood Cleese", 123456}
{"Michael Edward Palin", 987654}
```

See §7.3 for a more systematic technique for recognizing patterns in streams of characters (regular expression matching).

8.6 Formatting

The **iostream** library provides a large set of operations for controlling the format of input and output. The simplest formatting controls are called *manipulators* and are found in **<ios>**, **<istream>**, **<ostream>**, and **<iomanip>** (for manipulators that take arguments): For example, we can output integers as decimal (the default), octal, or hexadecimal numbers:

```
cout << 1234 << ',' << hex << 1234 << ',' << oct << 1234 << '\n';          // print 1234,4d2,2322
```

We can explicitly set the output format for floating-point numbers:

```
constexpr double d = 123.456;

cout << d << "; "                    // use the default format for d
     << scientific <<  d << "; "     // use 1.123e2 style format for d
     << hexfloat <<  d << "; "       // use hexadecimal notation for d
     << fixed << d << "; "           // use 123.456 style format for f
     << defaultfloat << d << '\n';   // use the default format for d
```

This produces:

```
123.456; 1.234560e+002; 0x1.edd2f2p+6; 123.456000; 123.456
```

Precision is an integer that determines the number of digits used to display a floating-point number:

- The *general* format (**defaultfloat**) lets the implementation choose a format that presents a value in the style that best preserves the value in the space available. The precision specifies the maximum number of digits.
- The *scientific* format (**scientific**) presents a value with one digit before a decimal point and an exponent. The precision specifies the maximum number of digits after the decimal point.
- The *fixed* format (**fixed**) presents a value as an integer part followed by a decimal point and a fractional part. The precision specifies the maximum number of digits after the decimal point.

Floating-point values are rounded rather than just truncated, and **precision()** doesn't affect integer output. For example:

```
cout.precision(8);
cout << 1234.56789 << ' ' << 1234.56789 << ' ' << 123456 << '\n';

cout.precision(4);
cout << 1234.56789 << ' ' << 1234.56789 << ' ' << 123456 << '\n';
```

This produces:

```
1234.5679 1234.5679 123456
1235 1235 123456
```

These manipulators as ‘‘sticky’’; that is, it persists for subsequent floating-point operations.

8.7 File Streams

In **<fstream>**, the standard library provides streams to and from a file:

- **ifstream**s for reading from a file
- **ofstream**s for writing to a file
- **fstream**s for reading from and writing to a file

For example:

```
ofstream ofs("target");                 // “o” for “output”
if (!ofs)
    error("couldn't open 'target' for writing");
```

Testing that a file stream has been properly opened is usually done by checking its state.

```
fstream ifs;                            // “i” for “input”
if (!ifs)
    error("couldn't open 'source' for reading");
```

Assuming that the tests succeeded, **ofs** can be used as an ordinary **ostream** (just like **cout**) and **ifs** can be used as an ordinary **istream** (just like **cin**).

File positioning and more detailed control of the way a file is opened is possible, but beyond the scope of this book.

8.8 String Streams

In **<sstream>**, the standard library provides streams to and from a **string**:

- **istringstream**s for reading from a **string**
- **ostringstream**s for writing to a **string**
- **stringstream**s for reading from and writing to a **string**.

For example:

```
void test()
{
    ostringstream oss;
```

```
        oss << "{temperature," << scientific << 123.4567890 << "}";
        cout << oss.str() << '\n';
}
```

The result from an **istringstream** can be read using **str()**. One common use of an **ostringstream** is to format before giving the resulting string to a GUI. Similarly, a string received from a GUI can be read using formatted input operations (§8.3) by putting it into an **istringstream**.

A **stringstream** can be used for both reading and writing. For example, we can define an operation that can convert any type with a string representation to another that also has a string representation:

```
template<typename Target =string, typename Source =string>
Target to(Source arg)           // convert Source to Target
{
    stringstream interpreter;
    Target result;

    if (!(interpreter << arg)                   // write arg into stream
        || !(interpreter >> result)             // read result from stream
        || !(interpreter >> std::ws).eof())     // stuff left in stream?
        throw runtime_error{"to<>() failed"};

    return result;
}
```

A function template argument needs to be explicitly mentioned only if it cannot be deduced or if there is no default, so we can write:

```
auto x1 = to<string,double>(1.2);   // very explicit (and verbose)
auto x2 = to<string>(1.2);          // Source is deduced to double
auto x3 = to<>(1.2);                // Target is defaulted to string; Source is deduced to double
auto x4 = to(1.2);                  // the <> is redundant;
                                    // Target is defaulted to string; Source is deduced to double
```

If all function template arguments are defaulted, the **<>** can be left out.

I consider this a good example of the generality and ease of use that can be achieved by a combination of language features and standard-library facilities.

8.9 Advice

[1] The material in this chapter roughly corresponds to what is described in much greater detail in Chapter 38 of [Stroustrup,2013].

[2] **iostream**s are type-safe, type-sensitive, and extensible; §8.1.

[3] Define **<<** and **>>** for user-defined types with values that have meaningful textual representations; §8.1, §8.2, §8.3.

[4] Use **cout** for normal output and **cerr** for errors; §8.1.

[5] There are **iostream**s for ordinary characters and wide characters, and you can define an **iostream** for any kind of character; §8.1.

[6] Binary I/O is supported; §8.1.
[7] There are standard **iostream**s for standard I/O streams, files, and **string**s; §8.2, §8.3, §8.7, §8.8.
[8] Chain **<<** operations for a terser notation; §8.2.
[9] Chain **>>** operations for a terser notation; §8.3.
[10] Input into **string**s does not overflow; §8.3.
[11] By default **>>** skips initial whitespace; §8.3.
[12] Use the stream state **fail** to handle potentially recoverable I/O errors; §8.4.
[13] You can define **<<** and **>>** operators for your own types; §8.5.
[14] You don't need to modify **istream** or **ostream** to add new **<<** and **>>** operators; §8.5.
[15] Use manipulators to control formatting; §8.6.
[16] **precision()** specifications apply to all following floating-point output operations; §8.6.
[17] Floating-point format specifications (e.g., **scientific**) apply to all following floating-point output operations; §8.6.
[18] **#include <ios>** when using standard manipulators; §8.6.
[19] **#include <iomanip>** when using standard manipulators taking arguments; §8.6.
[20] Don't try to copy a file stream.
[21] Remember to check that a file stream is attached to a file before using it; §8.7.
[22] Use **stringstream**s for in-memory formatting; §8.8.
[23] You can define conversions between any two types that both have string representation; §8.8.