15.1. Common Background of I/O Streams

Before going into details about stream classes, I briefly discuss the generally known aspects of streams to provide a common background. This section could be skipped by readers familiar with iostream basics.

15.1.1. Stream Objects

In C++, I/O is performed by using streams. A *stream* is a "*stream of data*" in which character sequences "*flow*." Following the principles of object orientation, a stream is an object with properties that are defined by a class. *Output* is interpreted as data flowing into a stream; *input* is interpreted as data flowing out of a stream. Global objects are predefined for the standard I/O channels.

15.1.2. Stream Classes

Just as there are different kinds of I/O — for example, input, output, and file access — there are different classes depending on the type of I/O. The following are the most important stream classes:

- **Class `istream`** defines input streams that can be used to read data.
- **Class `ostream`** defines output streams that can be used to write data.

Both classes are instantiations of the class templates `basic_istream<>` or `basic_ostream<>`, respectively, using `char` as the character type. In fact, the whole IOStream library does not depend on a specific character type. Instead, the character type used is a template argument for most of the classes in the IOStream library. This parametrization corresponds to the string classes and is used for internationalization (see also Chapter 16).

This section concentrates on output to and input from "narrow streams": streams dealing with `char` as the character type. Later in this chapter, the discussion is extended to streams that have other character types.

15.1.3. Global Stream Objects

The IOStream library defines several global objects of type `istream` and `ostream`. These objects correspond to the standard I/O channels:

- `cin`, of class `istream`, is the standard input channel used for user input. This stream corresponds to C's `stdin`. Normally, this stream is connected to the keyboard by the operating system.
- `cout`, of class `ostream`, is the standard output channel used for program output. This stream corresponds to C's `stdout`. Normally, this stream is connected to the monitor by the operating system.
- `cerr`, of class `ostream`, is the standard error channel used for all kinds of error messages. This stream corresponds to C's `stderr`. Normally, this stream is also connected to the monitor by the operating system. By default, `cerr` is not buffered.
- `clog`, of class `ostream`, is the standard logging channel. It has no C equivalent. By default, this stream is connected to the same destination as `cerr`, with the difference that output to `clog` is buffered.

The separation of "normal" output and error messages makes it possible to treat these two kinds of output differently when executing a program. For example, the normal output of a program can be redirected into a file while the error messages are still appearing on the console. Of course, this requires that the operating system support redirection of the standard I/O channels (most operating systems do). This separation of standard channels originates from the UNIX concept of I/O redirection.

15.1.4. Stream Operators

The shift operators `>>` for input and `<<` for output are overloaded for the corresponding stream classes. For this reason, the "shift operators" in C++ became the "I/O operators."[1] Using these operators, it is possible to chain multiple I/O operations.

[1] Because these operators insert characters into a stream or extract characters from a stream, some people also call the I/O operators *inserters* and *extractors*.

For example, for each iteration, the following loop reads two integers from the standard input as long as only integers are entered and writes them to the standard output:

Click here to view code image

```
int a, b;

// as long as input of a and b is successful
while (std::cin >> a >> b) {
    // output a and b
    std::cout << "a: " << a << " b: " << b << std::endl;
}
```

15.1.5. Manipulators

At the end of most output statements, a so-called manipulator is written:

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

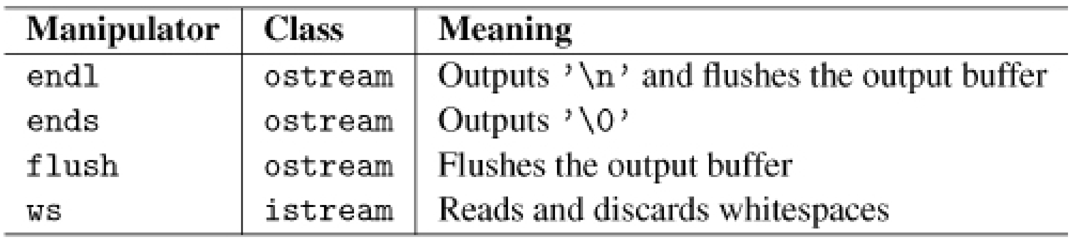
Manipulators are special objects that are used to, guess what, manipulate a stream. Often, manipulators change only the way input is interpreted or output is formatted, like the manipulators for the numeric bases `dec` , `hex` , and `oct` . Thus, manipulators for `ostream` s do not necessarily create output, and manipulators for `istream` s do not necessary consume input. But some manipulators do trigger some immediate action. For example, a manipulator can be used to flush the output buffer or to skip whitespace in the input buffer.

The manipulator `endl` means "end line" and does two things:

1. Outputs a newline (that is, the character `'\n'`)
2. Flushes the output buffer (forces a write of all buffered data for the given stream, using the stream method `flush()`)

The most important manipulators defined by the IOStream library are provided in Table 15.1. Section 15.6, page 774, discusses manipulators in more detail, including those that are defined in the IOStream library, and explains how to define your own manipulators.

Table 15.1. The IOStream Library's Most Important Manipulators



Manipulator	Class	Meaning
`endl`	`ostream`	Outputs `'\n'` and flushes the output buffer
`ends`	`ostream`	Outputs `'\0'`
`flush`	`ostream`	Flushes the output buffer
`ws`	`istream`	Reads and discards whitespaces

15.1.6. A Simple Example

The use of the stream classes is demonstrated by the following example. This program reads two floating-point values and outputs their product:

Click here to view code image

```
// io/io1.cpp

#include <cstdlib>
#include <iostream>
using namespace std;

int main()
{
    double x, y;           // operands

    // print header string
    cout << "Multiplication of two floating point values" << endl;

    // read first operand
    cout << "first operand:   ";
    if (! (cin >> x)) {
        // input error
        // => error message and exit program with error status
        cerr << "error while reading the first floating value"
             << endl;
        return EXIT_FAILURE;
    }

    // read second operand
    cout << "second operand: ";
    if (! (cin >> y)) {
        // input error
        // => error message and exit program with error status
        cerr << "error while reading the second floating value"
             << endl;
        return EXIT_FAILURE;
    }

    // print operands and result
```

```
        cout << x << " times " << y << " equals " << x * y << endl;
    }
```