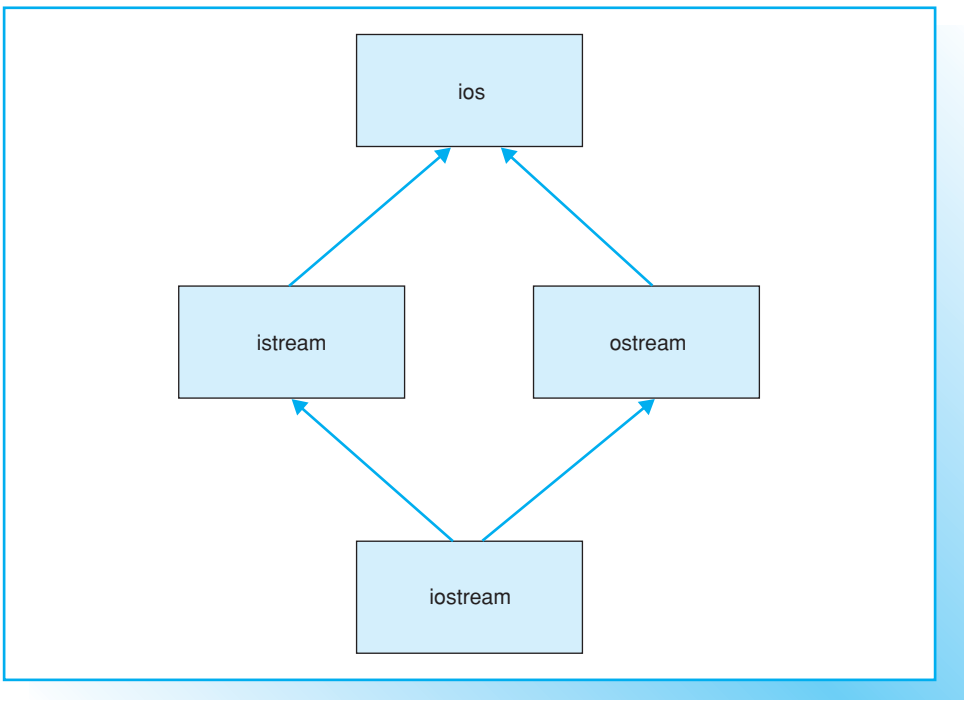
chapter 4

Input and Output with Streams

This chapter describes the use of streams for input and output, focusing on formatting techniques.

■ STREAMS

Stream classes for input and output



The four standard streams

- **`cin`** Object of class `istream` to control standard input
- **`cout`** Object of class `ostream` to control standard output
- **`cerr`** Object of class `ostream` to control unbuffered error output
- **`clog`** Object of class `ostream` to control buffered error output

I/O Stream Classes

During the development of C++ a new class-based input/output system was implemented. This gave rise to the *I/O stream classes*, which are now available in a library of their own, the so-called *iostream library*.

The diagram on the opposite page shows how a so-called class hierarchy develops due to inheritance. The class `ios` is the base class of all other stream classes. It contains the attributes and abilities common to all streams. Effectively, the `ios` class

- manages the connection to the physical data stream that writes your program's data to a file or outputs the data on screen
- contains the basic functions needed for formatting data. A number of flags that determine how character input is interpreted have been defined for this purpose.

The `istream` and `ostream` classes derived from `ios` form a user-friendly interface for stream manipulation. The `istream` class is used for reading streams and the `ostream` class is used for writing to streams. The operator `>>` is defined in `istream` and `<<` is defined in `ostream`, for example.

The `iostream` class is derived by multiple inheritance from `istream` and `ostream` and thus offers the functionality of both classes.

Further stream classes, a file management class, for example, are derived from the classes mentioned above. This allows the developer to use the techniques described for file manipulation. These classes, which also contain methods for opening and closing files, will be discussed in a later chapter.

Standard Streams

The streams `cin` and `cout`, which were mentioned earlier, are instances of the `istream` or `ostream` classes. When a program is launched these objects are automatically created to read *standard input* or write to *standard output*.

Standard input is normally the keyboard and standard output the screen. However, standard input and output can be redirected to files. In this case, data is not read from the keyboard but from a file, or data is not displayed on screen but written to a file.

The other two standard streams `cerr` and `clog` are used to display messages when errors occur. Error messages are displayed on screen even if standard output has been redirected to a file.

■ FORMATTING AND MANIPULATORS

Example: Calling a manipulator

Here the manipulator `showpos` is called.

```
cout << showpos << 123;    // Output:  +123
```

The above statement is equivalent to

```
cout.setf( ios::showpos);
cout << 123;
```

The other positive numbers are printed with their sign as well:

```
cout << 22;                // Output:  +22
```

The output of a positive sign can be canceled by the manipulator `noshowpos`:

```
cout << noshowpos << 123;  // Output:  123
```

The last statement is equivalent to

```
cout.unsetf( ios::showpos);
cout << 123;
```

HINTS

- The operators `>>` and `<<` format the input and/or output according to how the flags in the base class `ios` are set
- The manipulator `showpos` is a function that calls the method `cout.setf(ios::showpos);`, `ios::showpos` being the flag `showpos` belonging to the `ios` class
- Using manipulators is easier than directly accessing flags. For this reason, manipulators are described in the following section, whereas the methods `setf()` and `unsetf()` are used only under exceptional circumstances.
- Old compilers only supply some of the manipulators. In this case, you have to use the methods `setf()` and `unsetf()`.

Formatting

When reading keyboard input, a valid input format must be used to determine how input is to be interpreted. Similarly, screen output adheres to set of rules governing how, for example, floating-point numbers are displayed.

The stream classes `istream` and `ostream` offer various options for performing these tasks. For example, you can display a table of numeric values in a simple way.

In previous chapters we have looked at the `cin` and `cout` streams in statements such as:

```
cout << "Please enter a number: ";
cin  >> x;
```

The following sections systematically describe the abilities of the stream classes. This includes:

- the `>>` and `<<` operators for formatted input and output. These operators are defined for expressions with fundamental types—that is, for characters, boolean values, numbers and strings.
- manipulators, which can be inserted into the input or output stream. Manipulators can be used to generate formats for subsequent input/output. One manipulator that you are already familiar with is `endl`, which generates a line feed at the end of a line.
- other methods for determining or modifying the state of a stream and unformatted input and output.

Flags and Manipulators

Formatting flags defined in the parent class `ios` determine how characters are input or output. In general, flags are represented by individual bits within a special integral variable. For example, depending on whether a bit is set or not, a positive number can be output with or without a plus sign.

Each flag has a *default* setting. For example, integral numbers are output as decimals by default, and positive numbers are output without a plus sign.

It is possible to modify individual formatting flags. The methods `setf()` and `unsetf()` can be used for this purpose. However, the same effect can be achieved simply by using so-called *manipulators*, which are defined for all important flags. Manipulators are functions that can be inserted into the input or output stream and thus be called.

■ FORMATTED OUTPUT OF INTEGERS

Manipulators formatting integers

Manipulator	Effects
`oct`	Octal base
`hex`	Hexadecimal base
`dec`	Decimal base (by default)
`showpos`	Generates a + sign in non-negative numeric output.
`noshowpos`	Generates non-negative numeric output without a + sign (by default).
`uppercase`	Generates capital letters in hexadecimal output.
`nouppercase`	Generates lowercase letters in hexadecimal output (by default).

Sample program

```
// Reads integral decimal values and
// generates octal, decimal, and hexadecimal output.

#include <iostream>      // Declarations of cin, cout and
using namespace std;     // manipulators oct, hex, ...

int main()
{
   int number;
   cout << "Please enter an integer: ";
   cin >> number;
   cout << uppercase                 // for hex-digits
        << " octal  \t decimal  \t hexadecimal\n "
        << oct << number << "        \t "
        << dec << number << "        \t "
        << hex << number << endl;
   return 0;
}
```

☐ Formatting Options

The `<<` operator can output values of type `short`, `int`, `long` or a corresponding `unsigned` type. The following formatting options are available:

- define the numeric system in which to display the number: decimal, octal, or hexadecimal
- use capitals instead of small letters for hexadecimals
- display a sign for positive numbers.

In addition, the field width can be defined for the above types. The field width can also be defined for characters, strings, and floating-point numbers, and will be discussed in the following sections.

☐ Numeric System

Integral numbers are displayed as decimals by default. The manipulators `oct`, `hex`, and `dec` can be used for switching from and to decimal display mode.

Example:
```
cout << hex << 11;              // Output: b
```

Hexadecimals are displayed in small letters by default, that is, using `a`, `b`, ..., `f`. The manipulator `uppercase` allows you to use capitals.

Example:
```
cout << hex << uppercase << 11; //Output: B
```

The manipulator `nouppercase` returns the output format to small letters.

☐ Negative Numbers

When negative numbers are output as decimals, the output will always include a sign. You can use the `showpos` manipulator to output signed positive numbers.

Example:
```
cout << dec << showpos << 11; //Output: +11
```

You can use `noshowpos` to revert to the original display mode.

When *octal* or *hexadecimal* numbers are output, the bits of the number to be output are always interpreted as unsigned! In other words, the output shows the bit pattern of a number in octal or hexadecimal format.

Example:
```
cout << dec << -1 << "   " << hex << -1;
```

This statement causes the following output on a 32-bit system:

```
-1   ffffffff
```

■ FORMATTED OUTPUT OF FLOATING-POINT NUMBERS

Manipulators formatting floating-point numbers

Manipulator	Effects
`showpoint`	Generates a decimal point character shown in floating-point output. The number of digits after the decimal point corresponds to the used precision.
`noshowpoint`	Trailing zeroes after the decimal point are not printed. If there are no digits after the decimal point, the decimal point is not printed (by default).
`fixed`	Output in fixed point notation
`scientific`	Output in scientific notation
`setprecision (int n)`	Sets the precision to `n`.

Methods for precision

Manipulator	Effects
`int precision (int n);`	Sets the precision to `n`.
`int precision() const;`	Returns the used precision.

NOTE

The key word `const` within the prototype of `precision()` signifies that the method performs only read operations.

Sample program

```
#include <iostream>
using namespace std;
int main()
{
   double x = 12.0;
   cout.precision(2);              // Precision 2
   cout << " By default:    " << x << endl;
   cout << " showpoint:   " << showpoint   << x << endl;
   cout << " fixed:       " << fixed       << x << endl;
   cout << " scientific:  " << scientific  << x << endl;
   return 0;
}
```

☐ Standard Settings

Floating-points are displayed to six digits by default. Decimals are separated from the integral part of the number by a decimal point. Trailing zeroes behind the decimal point are not printed. If there are no digits after the decimal point, the decimal point is not printed (by default).

Examples:
```
cout << 1.0;        // Output: 1
cout << 1.234;      // Output: 1.234
cout << 1.234567;   // Output: 1.23457
```

The last statement shows that the seventh digit is not simply truncated but rounded. Very large and very small numbers are displayed in *exponential notation*.

Example:
```
cout << 1234567.8; // Output: 1.23457e+06
```

☐ Formatting

The standard settings can be modified in several ways. You can

- change the precision, i.e. the number of digits to be output
- force output of the decimal point and trailing zeroes
- stipulate the display mode (fixed point or exponential).

Both the manipulator `setprecision()` and the method `precision()` can be used to redefine precision to be used.

Example:
```
cout << setprecision(3);  // Precision: 3
// or:  cout.precision(3);
cout  << 12.34;           // Output: 12.3
```

Note that the header file `iomanip` must be included when using the manipulator `setprecision()`. This also applies to all standard manipulators called with at least one argument.

The manipulator `showpoint` outputs the decimal point and trailing zeroes. The number of digits being output (e.g. 6) equals the current precision.

Example:
```
cout << showpoint << 1.0; // Output: 1.00000
```

However, *fixed point* output with a predetermined number of decimal places is often more useful. In this case, you can use the `fixed` manipulator with the precision defining the number of decimal places. The default value of 6 is assumed in the following example.

Example:
```
cout << fixed << 66.0;   // Output: 66.000000
```

In contrast, you can use the `scientific` manipulator to specify that floating-point numbers are output as exponential expressions.

■ OUTPUT IN FIELDS

Element functions for output in fields

Method	Effects
`int width() const;`	Returns the minimum field width used
`int width(int n);`	Sets the minimum field width to `n`
`int fill() const;`	Returns the fill character used
`int fill(int ch);`	Sets the fill character to `ch`

Manipulators for output in fields

Manipulator	Effects
`setw(int n)`	Sets the minimum field width to `n`
`setfill(int ch)`	Sets the fill character to `ch`
`left`	Left-aligns output in fields
`right`	Right-aligns output in fields
`internal`	Left-aligns output of the sign and right-aligns output of the numeric value

NOTE

The manipulators `setw()` and `setfill()` are declared in the header file `iomanip`.

Examples

```
#include <iostream>          // Obligatory
#include <iomanip>           // declarations
using namespace std;

1st Example: cout << '|' << setw(6) << 'X' << '|';
Output:      |     X|          // Field width 6
2nd Example: cout << fixed << setprecision(2)
                  << setw(10) << 123.4 << endl
                  << "1234567890" << endl;
Output:         123.40          // Field width 10
           1234567890
```

The << operator can be used to generate formatted output in fields. You can

- specify the *field width*
- set the alignment of the output to right- or left-justified
- specify a *fill-character* with which to fill the field.

☐ Field Width

The field width is the number of characters that can be written to a field. If the output string is larger than the field width, the output is not truncated but the field is extended. The output will always contain at least the number of digits specified as the field width.

You can either use the `width()` method or the `setw()` manipulator to define field width.

Example:
```
cout.width(6);     // or:  cout << setw(6);
```

One special attribute of the field width is the fact that this value is non-permanent: the field width specified applies to the next output only, as is illustrated by the examples on the opposite page. The first example outputs the character `'X'` to a field with width of 6, but does not output the `'|'` character.

The default field width is 0. You can also use the `width()` method to get the current field width. To do so, call `width()` without any other arguments.

Example:
```
int fieldwidth = cout.width();
```

☐ Fill Characters and Alignment

If a field is larger than the string you need to output, blanks are used by default to fill the field. You can either use the `fill()` method or the `setfill()` manipulator to specify another fill character.

Example:
```
cout << setfill('*') << setw(5) << 12;
// Output: ***12
```

The fill character applies until another character is defined.

As the previous example shows, output to fields is normally right-aligned. The other options available are left-aligned and internal, which can be set by using the manipulators `left` and `internal`. The manipulator `internal` left-justifies the sign and right-justifies the number within a field.

Example:
```
cout.width(6); cout.fill('0');
cout << internal << -123; // Output: -00123
```

■ OUTPUT OF CHARACTERS, STRINGS, AND BOOLEAN VALUES

Sample program

```
// Enters a character and outputs its
// octal, decimal, and hexadecimal code.

#include <iostream>     // Declaration of cin, cout
#include <iomanip>      // For manipulators being called
                        // with arguments.
#include <string>
using namespace std;

int main()
{
   int number = ' ';

   cout << "The white space code is as follows: "
        << number << endl;

   char ch;
   string prompt =
          "\nPlease enter a character followed by "
                                        " <return>: ";

   cout << prompt;

   cin >> ch;                       // Read a character
   number = ch;

   cout << "The character " << ch
        << " has code" << number << endl;

   cout << uppercase                // For hex-digits
        << "     octal  decimal  hexadecimal\n "
        << oct << setw(8) << number
        << dec << setw(8) << number
        << hex << setw(8) << number << endl;

   return 0;
}
```

☐ Outputting Characters and Character Codes

The >> operator interprets a number of type `char` as the character code and outputs the corresponding character:

Example:
```
char ch = '0';
cout << ch << ' ' << 'A';
// Outputs three characters: 0 A
```

It is also possible to output the character code for a character. In this case the character code is stored in an `int` variable and the variable is then output.

Example:
```
int code = '0';
cout << code;          // Output: 48
```

The `'0'` character is represented by ASCII Code `48`. The program on the opposite page contains further examples.

☐ Outputting Strings

You can use the >> operator both to output string literals, such as `"Hello"`, and string variables, as was illustrated in previous examples. As in the case of other types, strings can be positioned within output fields.

Example:
```
string s("spring flowers ");
cout << left            // Left-aligned
     << setfill('?')    // Fill character ?
     << setw(20) << s ; // Field width 20
```

This example outputs the string `"spring  flowers??????"`. The manipulator `right` can be used to right-justify the output within the field.

☐ Outputting Boolean Values

By default the << operator outputs boolean values as integers, with the value `0` representing `false` and `1` `true`. If you need to output the strings `true` or `false` instead, the flag `ios::boolalpha` must be set. To do so, use either the `setf()` method or the manipulator `boolalpha`.

Example:
```
bool ok = true;
cout << ok << endl                  // 1
     << boolalpha << ok << endl;    // true
```

You can revert this setting using the `noboolalpha` manipulator.

■ FORMATTED INPUT

Sample program

```
// Inputs an article label and a price

#include <iostream>    // Declarations of cin, cout,...
#include <iomanip>     // Manipulator setw()
#include <string>
using namespace std;

int main()
{
   string label;
   double price;

   cout << "\nPlease enter an article label: ";

   // Input the label (15 characters maximum):
   cin >> setw(16);        // or:  cin.width(16);
   cin >> label;

   cin.sync();    // Clears the buffer and resets
   cin.clear();   // any error flags that may be set

   cout << "\nEnter the price of the article: ";
   cin >> price;           // Input the price

   // Controlling output:
   cout << fixed << setprecision(2)
        << "\nArticle:"
        << "\n  Label:  " << label
        << "\n  Price:  " << price << endl;

   // ... The program to be continued

   return 0;
}
```

NOTE

The input buffer is cleared and error flags are reset by calling the `sync()` and `clear()` methods. This ensures that the program will wait for new input for the price, even if more than 15 characters have been entered for the label.

The >> operator, which belongs to the `istream` class, takes the current number base and field width flags into account when reading input:

- the number base specifies whether an integer will be read as a decimal, octal, or hexadecimal
- the field width specifies the maximum number of characters to be read for a string.

When reading from standard input, `cin` is buffered by lines. Keyboard input is thus not read until confirmed by pressing the <Return> key. This allows the user to press the backspace key and correct any input errors, provided the return key has not been pressed. Input is displayed on screen by default.

☐ Input Fields

The >> operator will normally read the next *input* field, convert the input by reference to the type of the supplied variable, and write the result to the variable. Any white space characters (such as blanks, tabs, and new lines) are ignored by default.

Example:
```
char ch;
cin >> ch;          // Enter a character
```

When the following keys are pressed

```
<return> <tab> <blank> <X> <return>
```

the character `'X'` is stored in the variable `ch`.

An input field is terminated by the first white space character or by the first character that cannot be processed.

Example:
```
int i;
cin >> i;
```

Typing `123FF<Return>` stores the decimal value `123` in the variable `i`. However, the characters that follow, `FF` and the newline character, remain in the input buffer and will be read first during the next read operation.

When reading strings, only one word is read since the first white space character will begin a new input field.

Example:
```
string city;
cin >> city;      // To read just one word!
```

If `Lao Kai` is input, only `Lao` will be written to the `city` string. The number of characters to be read can also be limited by specifying the field width. For a given field width of `n`, a maximum of `n-1` characters will be read, as one byte is required for the null character. Any initial white space will be ignored. The program on the opposite page illustrates this point and also shows how to clear the input buffer.

■ FORMATTED INPUT OF NUMBERS

Sample program

```
// Enter hexadecimal digits and a floating-point number
//
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
   int number = 0;

   cout << "\nEnter a hexadecimal number: "
        << endl;
   cin >> hex >> number;       // Input hex-number

   cout << "Your decimal input: " << number << endl;

   // If an invalid input occurred:
   cin.sync();                 // Clears the buffer
   cin.clear();                // Reset error flags

   double x1 = 0.0, x2 = 0.0;

   cout << "\nNow enter two floating-point values: "
        << endl;

   cout << "1. number: ";
   cin  >> x1;                          // Read first number
   cout << "2. number: ";
   cin  >> x2;                          // Read second number

   cout << fixed << setprecision(2)
        << "\nThe sum of both numbers:    "
        << setw(10) << x1 + x2 << endl;

   cout << "\nThe product of both numbers: "
        << setw(10) << x1 * x2 << endl;

   return 0;
}
```

☐ Inputting Integers

You can use the `hex`, `oct`, and `dec` manipulators to stipulate that any character sequence input is to processed as a hexadecimal, octal, or decimal number.

Example:
```
int n;
cin >> oct >> n;
```

An input value of `10` will be interpreted as an octal, which corresponds to a decimal value of `8`.

Example:
```
cin >> hex >> n;
```

Here, any input will be interpreted as a hexadecimal, enabling input such as `f0a` or `-F7`.

☐ Inputting Floating-Point Numbers

The `>>` operator interprets any input as a decimal floating-point number if the variable is a floating-point type, i.e. `float`, `double`, or `long double`. The floating-point number can be entered in fixed point or exponential notation.

Example:
```
double x;
cin >> x;
```

The character input is converted to a `double` value in this case. Input, such as `123`, `-22.0`, or `3e10` is valid.

☐ Input Errors

But what happens if the input does not match the type of variable defined?

Example:
```
int i, j;    cin >> i >> j;
```

Given input of `1A5` the digit `1` will be stored in the variable `i`. The next input field begins with `A`. But since a decimal input type is required, the input sequence will not be processed beyond the letter `A`. If, as in our example, no type conversion is performed, the variable is not written to and an internal error flag is raised.

It normally makes more sense to read numerical values individually, and clear the input buffer and any error flags that may have been set after each entry.

Chapter 6, "Control Flow," and Chapter 28, "Exception Handling," show how a program can react to input errors.

■ UNFORMATTED INPUT/OUTPUT

Sample program

```
//  Reads a text with the operator >>
//  and the function getline().

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

string header =
"   --- Demonstrates Unformatted Input ---";

int main()
{
   string word, rest;

   cout << header
        << "\n\nPress <return> to go on" << endl;


   cin.get();                    // Read the new line
                                 // without saving.

   cout << "\nPlease enter a sentence with several words!"
        << "\nEnd with <!> and <return>."
        << endl;

   cin >> word;                  // Read the first word
   getline( cin, rest, '!');     // and the remaining text
                                 // up to the character !

   cout << "\nThe first word:   " << word
        << "\nRemaining text: " << rest << endl;

   return 0;
}
```

NOTE

1. A text of more than one line can be entered.
2. The sample program requires that at least one word and a following white space are entered.

Unformatted input and output does not use fields, and any formatting flags that have been set are ignored. The bytes read from a stream are passed to the program "as is." More specifically, you should be aware that any white space characters preceding the input will be processed.

☐ Reading and Writing Characters

You can use the methods `get()` and `put()` to read or write single characters. The `get()` method reads the next character from a stream and stores it in the given `char` variable.

Example:
```
char ch;
cin.get(ch);
```

If the character is a white space character, such as a newline, it will still be stored in the `ch` variable. To prevent this from happening you can use

```
cin >> ch;
```

to read the first non-white space character.

The `get()` method can also be called without any arguments. In this case, `get()` returns the character code of type `int`.

Example:
```
int c = cin.get();
```

The `put()` method can be used for unformatted output of a character. The character to be output is passed to `put()` as an argument.

Example:
```
cout.put('A');
```

This statement is equivalent to `cout << 'A';`, where the field width is undefined or has been set to 1.

☐ Reading a Line

The `>>` operator can only be used to read one word into a string. If you need to read a whole line of text, you can use the global function `getline()`, which was introduced earlier in this chapter.

Example:
```
getline(cin, text);
```

This statement reads characters from `cin` and stores them in the string variable `text` until a new line character occurs. However, you can specify a different delimiting character by passing the character to the `getline()` function as a third argument.

Example:
```
getline(cin, s, '.');
```

The delimiting character is read, but not stored in the string. Any characters subsequent to the first period will remain in the input buffer of the stream.