How to work with output streams

In chapter 2, you learned the basic skills for displaying output to the console. Now, you'll learn some additional skills for working with output streams. These skills include how to display data in columns that are left or right justified, and how to control the formatting of floating-point numbers.

An introduction to stream manipulators

Figure 5-6 starts by summarizing the most common *stream manipulators*, which you can use with the stream insertion operator (<<) to format output streams. These manipulators are available from the iostream and iomanip header files, and they include the endl manipulator that you learned about in chapter 2 as well as six more manipulators that you'll learn about in this chapter.

As you review these operators, note that the endl manipulator adds a newline character (\n) to the stream and flushes the buffer. If you manually add a newline character (\n) to a stream, however, the buffer isn't flushed. When you're working with large files, flushing the buffer too often can degrade performance. As a result, if you're worried about performance, you can use the newline character (\n) instead of the endl manipulator whenever it isn't necessary to flush the buffer.

How to specify the width of a column

If you need to display data in columns, you can use the tab character (\t) to separate columns. However, using tab characters doesn't always work correctly, and it doesn't allow you to specify the width of the column. In figure 5-6, for instance, the first example uses tab characters to align two columns of data. Here, the header for each column isn't aligned with the data for the column.

To solve this problem, you can use the setw() manipulator as shown in the second example. Here, the setw() manipulator specifies that each column should be 10 characters wide. In addition, when you use the setw() manipulator, it uses right justification by default, which is often what you want when displaying numbers in columns.

When you use the setw() manipulator, you must code it before each data element. In this example, the output consists of two columns and four rows. As a result, you must code the setw() manipulator eight times.

Common stream manipulators

Manipulator	Header	Description
`endl`	`iostream`	Adds a newline character to the stream and flushes the buffer.
`setw(n)`	`iomanip`	Sets the width of the column to the specified number of characters.
`left`	`iostream`	Positions fill characters to the left of any output.
`right`	`iostream`	Positions fill characters to the right of any output.
`setprecision(n)`	`iomanip`	Sets the precision of floating-point numbers to the specified number.
`fixed`	`iostream`	Formats floating-point numbers in fixed-point notation instead of scientific notation.
`showpoint`	`iostream`	Makes decimals always show in floating-point numbers.

How to include the iomanip header file

```
#include <iomanip>
```

How to align columns with tabs

```
cout << "PREVIOUS" << '\t' << "CURRENT" << endl;
cout << 305 << '\t' << 234 << endl;
cout << 5058 << '\t' << 5084 << endl;
cout << 10647 << '\t' << 10759 << endl;
```

The output

```
PREVIOUS        CURRENT
305     234
5058    5084
10647   10759
```

How to specify the width of a column

```
cout << setw(10) << "PREVIOUS" << setw(10) << "CURRENT" << endl;
cout << setw(10) << 305 << setw(10) << 234 << endl;
cout << setw(10) << 5058 << setw(10) << 5084 << endl;
cout << setw(10) << 10647 << setw(10) << 10759 << endl;
```

The output

```
  PREVIOUS   CURRENT
       305       234
      5058      5084
     10647     10759
```

Description

- You can use *stream manipulators* with the stream insertion (<<) operator to format output streams.
- Both the endl manipulator and the '\n' special character add a new line to a stream, but endl also flushes the buffer. This can degrade performance when you're working with files.
- When you use the setw() manipulator, the column uses right justification by default. This is often what you want when displaying numbers in columns.

Figure 5-6 How to specify the width of a column

How to right or left justify columns

As you learned in the previous figure, the setw() manipulator uses right justification by default. However, when using the setw() manipulator, you can use the left and right manipulators to left and right justify the data in a column as shown in figure 5-7.

When aligning data in columns, it's common to use left justification for strings and right justification for numbers. To illustrate, the first example uses the default right justification for two columns of strings that are 14 characters wide. As you can see, this data isn't as easy to read as it could be.

To improve the appearance and readability of this data, the second example specifies left justification for these columns. Since the left and right manipulators are *sticky*, this example only sets the justification to left once. Then, the rest of the code is the same as in the first example.

Often, you need to use left justification for some columns and right justification for others. For this type of data, you need to code the left and right manipulators each time you want to change justification. For instance, the third example switches the justification for every column to create a first column that's left justified and a second column that's right justified. Here, the values in the second column are numbers that contain two decimal places, so aligning them at the right causes the decimal points to be aligned.

Text data displayed with the default right justification

```
cout << setw(14) << "FIRST NAME" << setw(14) << "LAST NAME" << endl;
cout << setw(14) << "Grace" << setw(14) << "Hopper" << endl;
cout << setw(14) << "Bjarne" << setw(14) << "Stroustrup" << endl;
```

The output

```
     FIRST NAME      LAST NAME
          Grace         Hopper
         Bjarne     Stroustrup
```

How to left-justify the text with the left manipulator

```
cout << left;
cout << setw(14) << "FIRST NAME" << setw(14) << "LAST NAME" << endl;
cout << setw(14) << "Grace" << setw(14) << "Hopper" << endl;
cout << setw(14) << "Bjarne" << setw(14) << "Stroustrup" << endl;
```

The output

```
FIRST NAME    LAST NAME
Grace         Hopper
Bjarne        Stroustrup
```

Data that uses left and right justification

```
cout << left << setw(14) << "Total:"
     << right << setw(8) << 199.99 << endl;
cout << left << setw(14) << "Taxes:"
     << right << setw(8) << 14.77 << endl;
cout << left << setw(14) << "Grand total:"
     << right << setw(8) << 185.22 << endl;
```

The output

```
Total:          199.99
Taxes:           14.77
Grand total:    185.22
```

Description

- When using the setw() manipulator, you can use the left and right manipulators to left and right justify the data in a column.
- It's common to use left justification for text and right justification for numbers.
- The left and right manipulators are *sticky*. This means that once you set them, they stay in effect until you change them or the program ends.

Figure 5-7 How to right or left justify a column

How to format floating-point numbers

Figure 5-8 shows how to format floating-point numbers. To start, you should know that, by default, a floating-point number is displayed with no more than six *significant digits*. A significant digit is a digit that carries meaning about the accuracy of a measurement. Significant digits include zeros when they're between other non-zero digits, but they don't include leading zeros or trailing zeros. For example, 109900 has four significant digits (1099). That's because the trailing zeros aren't necessary when the number is displayed in scientific notation (1.099×10^5). Conversely, 0.001099 has the same four significant digits (1099). That's because the leading zeros aren't necessary when the number is displayed in scientific notation (1.099×10^{-3}).

One way to format floating-point numbers is to round them to a specified number of significant digits. To do that, you can use the setprecision() manipulator. The first example shows how this works. To start, it defines and initializes two double values that both have 7 significant digits. Then, it displays these numbers with three significant digits and then five significant digits. If you didn't use the setprecision() function here, both values would be displayed with the default of six significant digits.

In this example, the second number is very large. As a result, C++ uses scientific notation to display it. However, the first number is neither very large nor very small. As a result, C++ uses standard notation to display it.

The second example shows how to use the fixed manipulator to display a specified number of decimal places using *fixed-point notation*. Here, the fixed manipulator indicates that the number should have a fixed number of decimal places. Then, the setprecision() manipulator indicates that the number should have 2 decimal places. As a result, C++ displays all of the numbers in this example with two decimal places. To do that, it rounds the first number, and it adds trailing zeros to the next two numbers.

The third example shows how to use the showpoint manipulator to force the display of trailing zeros. Here, the precision is set to 8. By default, C++ doesn't display trailing zeros that don't add meaning to the number unless you use the fixed manipulator. As a result, this code begins by displaying the double value of 4500.0 without any trailing zeros. However, this code continues by using the showpoint manipulator to display 8 digits, four to the left of the decimal point, and four to the right of it.

When you work with the setprecision(), fixed, and showpoint manipulators, you need to remember that they're sticky. As a result, they stay in effect until you change them or the program ends.

How to set the number of significant digits

```
double d1 = 1.012345;
double d2 = 101234500000;
cout << setprecision(3) << "3 significant digits\n"
     << d1 << endl
     << d2 << endl << endl
     << setprecision(5) << "5 significant digits\n"
     << d1 << endl
     << d2 << endl;
```

The output

```
3 significant digits
1.01
1.01e+11

5 significant digits
1.0123
1.0123e+11
```

How to set the number of decimal places

```
cout << fixed << setprecision(2)
     << 10.125 << endl
     << 19.5 << endl
     << 42.0 << endl;
```

The output

```
10.13
19.50
42.00
```

How to force trailing zeros

```
double d3 = 4500.0;
cout << setprecision(8)
     << d3 << endl
     << showpoint
     << d3 << endl;
```

The output

```
4500
4500.0000
```

Description

- The setprecision() manipulator rounds floating-point numbers to the specified number of *significant digits*, which are the digits that carry meaning about the accuracy of a measurement.
- The fixed manipulator forces the output of floating-point numbers to be in *fixed-point notation*, rather than the default scientific notation.
- The showpoint manipulator forces the display of trailing zeros.
- The setprecision(), fixed, and showpoint manipulators are sticky. As a result, they stay in effect until you change them or the program ends.

Figure 5-8 How to format floating-point numbers