How to: Format Output and Use Streams B

B.1 Formatting Output

Most of the programs we've shown have generated unformatted output. We used the stream **manipulator** `setw` in *windchill.cpp*, Program 5.8 (and some other programs), to force column-aligned output, but we have concentrated more on program design and development than on making output look good.

In addition to well-formatted code, good programs generate well-formatted output. The arrangement of the output aids the program user in interpreting data. However, it is altogether too easy for a programmer to spend an inordinate amount of time formatting output trying to make it "pretty." The objective of this book is to present broad programming concepts, which include documenting code and formatting output. You should strike a balance between the objectives of producing working programs and writing programs so that users can understand both the program and the output.

There are two methods for altering an output stream to change the format of values that are inserted into the stream: using stream member functions and using an object called a manipulator, accessible via the include file `<iomanip>` (or `<iomanip.h>`). Formatting functions and manipulators are summarized in Tables B.1 and B.2. In general it's much easier to use a manipulator than the corresponding stream formatting member function. Three programs illustrate how the output functions and manipulators change a stream so that strings and numbers are formatted according to several criteria.

Using Flags. Stroustrup [Str97] calls using flags "a time-honored if somewhat oldfashioned technique." Sticking to manipulators, which don't use flags, will make your life simpler than using the flag-based member functions of the `ostream` hierarchy. A flag is conceptually either on or off. Rather than using several `bool` variables, flags are normally packed as individual bits in one int. For example, since eight flags can be stored in an eight-bit number; one eight-bit number represents 256 different combinations of flags being on or off.

B.1.1 General and Floating-Point Formatting

Table B.1 shows the stream member functions that take parameters and their corresponding manipulators. All stream formatting functions except for `width` are *persistent*; once applied to the stream they stay in effect until they're removed. The `width` function,

Table B.1 Parameterized Stream Formatting Functions and Manipulators

Stream Function	Manipulator	Brief Description
`width(int w)`	`setw(int w)`	output field-width
`precision(int p)`	`setprecision(int p)`	# of digits
`fill(), fill(int)`	`setfill(int)`	pad/fill character
`setf(int )`	`setiosflags`	add flags
`unsetf(int)`	`resetiosflags`	remove flags
`flags(), flags(int)`		read, set flags

and it's corresponding manipulator `setw`, affect only the next string or number output. The functions in Table B.1 are used in Program B.1 as part of Section B.1.1. They're summarized below.

- *width(int n)* sets the field-width of the next string or numeric output to the specified width. Output is padded with blanks (see *fill*) as needed. Output that requires a width larger than `n` isn't truncated, it overflows the specified width.
- *precision(int n)* sets the number of digits that appear in floating-point output. This is the number of digits to the right of the decimal point in either `fixed` or `scientific` format, and the total number of digits otherwise, (see Program B.1 for examples). Values are rounded, not truncated.
- *fill(int n)* sets the fill character to `n` and returns the old fill value. Without a parameter `fill()` returns the current fill value.
- *setf(int n)* sets the flag(s) specified by `n`, without affecting other flag values. Similarly, `unsetf` unsets one or more flags. More than one flag can be specified by using bitwise-or as shown in Program B.3.
- *flags(int n)* sets the flags to `n`, the only flags set are those in `n`, and returns the old flags. In contrast, `setf` leaves other flags unaffected. Without a parameter `flags` returns the current flags. These functions are demonstrated in Program B.3.

B.1.2 Manipulators

Most of the flags that can be set using the stream member functions `setf`, `flags`, and `unsetf` are given in Table B.2. These flags are very cumbersome to use since some require specifying an additional parameter when using `setf`. For example, the following statements set left justification (the default justification is right) then generate as output `'1.23  '` with two spaces of padding/fill.

```
cout.setf(ios_base::left,  ios_base::adjustfield);
cout << "'"; cout.width(6);
cout << 1.23 << "'" << endl;
```

It's much simpler to use a manipulator; the statement below has the same effect.

```
cout << left << "'" << setw(6) << 1.23 << "'" << endl;
```

Table B.2 Stream formatting flags and corresponding manipulators. The flags are used with the `setf` stream member function; all flags are static constants in the class `ios_base`, called `ios` in earlier versions of C++. See Program B.2.

Stream Flags	**flag option**	**Manipulator**
hex, oct, dec	`io_base::basefield`	hex, oct, dec
left, right	`ios_base::adjustfield`	left, right
fixed, scientific	`ios_base::floatfield`	fixed, scientific
showbase	*none*	showbase, noshowbase
showpoint	*none*	showpoint, noshowpoint
boolalpha	*none*	boolalpha, noboolalpha
showpos	*none*	showpos, noshowpos

The flags and manipulators in Table B.2 are summarized here. They are used in the programs that follow.

- *hex, oct, dec* set the base of numeric output to 16, 8, and 10, respectively. The default base is 10. If `showbase` is specified octal numbers are preceded by a zero and hexadecimal numbers are preceded by `0x`, see Program B.3.
- *left, right* set the justification of string and numeric output. These don't have a visible effect unless the output requires a width smaller than the default width, six, or than the width specified by using `setw/width`, then fill characters are added to the right and left, respectively (left justification means adding fill characters to the right.)
- *showbase, showpoint, showpos*, respectively show what base is in effect, show a decimal point, and show a leading plus sign. Without `showpoint`, the value 70.0 is displayed as 70, regardless of the precision value. With `showpoint` as many zeros are shown as set by `precision`.
- *boolalpha* makes `true` and `false` display as those strings rather than 1 and 0. See Program B.3 for an example.

Precision and Justification for Floating-Point Values Using Manipulators. Formatted output for floating-point numbers is shown in Program B.1, *formatdemo.cpp*.

Program B.1 formatdemo.cpp

```
#include <iostream>
#include <iomanip>
#include <cmath>
using namespace std;

// formatting using manipulators
```

```
int main()
{
    const double CENTIPI = 100 * acos(−1); // arccos(-1) = PI
    const int MAX = 10;
    const int TAB = 15;
    int k;

    cout << "default setting " << CENTIPI << ", with setprecision(4), "
         << setprecision(4) << CENTIPI << endl;

    cout << "\nfixed floating-point, precision varies, fixed/scientific\n" << endl;
    for(k=0; k < MAX; k++)
    {  cout << left << "pre. " << k << "\t" << setprecision(k) << setw(TAB)
            << fixed << CENTIPI <<  scientific  << "\t" << CENTIPI << endl;
    }

    cout << endl << "width and justification vary, fixed, precision 2\n" << endl;
    cout << setprecision(2) << fixed;
    for(k=3; k < MAX+3; k++)
    {  cout << "wid. " << k << "\t+" << left << setw(k) << CENTIPI << right
            << "+\t\t-" << setw(k) << CENTIPI << "-" << endl;
    }

    cout << endl << "repeated, fill char = @\n" << endl;
    cout << setfill('@');
    for(k=3; k < MAX+3; k++)
    {  cout << "wid. " << k << "\t+" << left << setw(k) << CENTIPI << right
            << "+\t\t-" << setw(k) << CENTIPI << "-" << endl;
    }


    return 0;
}
```

formatdemo.cpp

The manipulator `setw` affects the next numeric output only, other manipulators are persistent and last until changed, for example, `precision`, `left`, and `right`. See Table B.2 for descriptions of manipulators. The manipulator `precision` rounds floating-point values rather than truncating them. When floating-point values are printed using either `fixed` or `scientific`, the precision is the number of decimal digits, otherwise (see the first line of output) the precision is the total number of digits. The default precision is six, as shown on the first line of output. The justification is set to `left` in the first loop, but varies in the subsequent output.

OUTPUT

```
prompt> formatdemo
default setting 314.159, with setprecision(4), 314.2

fixed floating-point, precision varies, fixed/scientific

pre. 0  314             3.141593e+002
pre. 1  314.2           3.1e+002
pre. 2  314.16          3.14e+002
pre. 3  314.159         3.142e+002
pre. 4  314.1593        3.1416e+002
pre. 5  314.15927       3.14159e+002
pre. 6  314.159265      3.141593e+002
pre. 7  314.1592654     3.1415927e+002
pre. 8  314.15926536    3.14159265e+002
pre. 9  314.159265359   3.141592654e+002

width and justification vary, fixed, precision 2

wid. 3  +314.16+                -314.16-
wid. 4  +314.16+                -314.16-
wid. 5  +314.16+                -314.16-
wid. 6  +314.16+                -314.16-
wid. 7  +314.16 +               - 314.16-
wid. 8  +314.16  +              -  314.16-
wid. 9  +314.16   +             -   314.16-
wid. 10 +314.16    +            -    314.16-
wid. 11 +314.16     +           -     314.16-
wid. 12 +314.16      +          -      314.16-

repeated, fill char = @

wid. 3  +314.16+                -314.16-
wid. 4  +314.16+                -314.16-
wid. 5  +314.16+                -314.16-
wid. 6  +314.16+                -314.16-
wid. 7  +314.16@+               -@314.16-
wid. 8  +314.16@@+              -@@314.16-
wid. 9  +314.16@@@+             -@@@314.16-
wid. 10 +314.16@@@@+            -@@@@314.16-
wid. 11 +314.16@@@@@+           -@@@@@314.16-
wid. 12 +314.16@@@@@@+          -@@@@@@314.16-
```

Formatting Using Stream Member Functions. Program B.2 demonstrates some of the same formatting features shown in Program B.1, but using stream member functions instead of manipulators. As you can see, using manipulators is much simpler.

Program B.2 streamflags.cpp

```
#include <iostream>
#include <cmath>
using namespace std;

int main()
{
    const double PI = acos(−1);  // arccos(-1) = PI radians
    const int MAX = 10;          // max precision used in demo
    int k;
    // set right justified, fixed floating format

    cout.setf(ios_base::right, ios_base::adjustfield);
    cout.setf(ios_base::fixed, ios_base::floatfield);

    cout << "fixed, right justfied, width 10, precision varies\n" << endl;

    for(k=0; k < MAX; k++)
    {   cout.precision(k);
        cout << k << "\t+";
        cout.width(MAX);
        cout << PI << "+" << endl;
    }

    // use different fill characters

    int fillc = cout.fill();
    cout.precision(2);
    cout << "\nshow fill char, precision is 2\n" << endl;
    for(k='a'; k <= 'd'; k++)
    {   cout << "old fill = '" << char(fillc) << "' +";
        cout.width(MAX);
        cout.fill(k);
        cout << PI << "+" << endl;
        fillc = cout.fill();
    }

    return 0;
}
```

streamflags.cpp

OUTPUT

```
prompt> streamflags
fixed, right justified, width 10, precision varies

0       +         3+
1       +       3.1+
2       +      3.14+
3       +     3.142+
4       +    3.1416+
5       +   3.14159+
6       +  3.141593+
7       + 3.1415927+
8       +3.14159265+
9       +3.141592654+

show fill char, precision is 2

old fill = ' ' +aaaaaa3.14+
old fill = 'a' +bbbbbb3.14+
old fill = 'b' +cccccc3.14+
old fill = 'c' +dddddd3.14+
```

Using Flags as Parameters. Program B.3 shows how to pass format flags as parameters. The stream member function `flags` returns the current flags, but also sets the flags to the value of its parameter as shown in the function `output`. Flags can be combined using the bitwise or operator, `operator |`, as shown in `main`. Each flag as a bit is one or zero. The bitwise or operation corresponds to a boolean or, but uses bits instead. In Program B.3, the result of combining the bits with or is a single number in which both flags are set.

Program B.3 formatparams.cpp

```
#include <iostream>
#include <iomanip>
using namespace std;

void output(ostream& out, ios_base::fmtflags flags)
// post: print using flags, restore old flags
{
    ios_base::fmtflags oldflags = out.flags(flags);

    out << "oldflags: " << oldflags << "\tnew: "
```

```
        << out.flags() << "\t";

    out << 12.47 << "\t" << true << "\t"
        << 99.0  << "\t" << 255 << endl;

    out.flags(oldflags);  // restore as before
}

int main()
{
    output(cout, cout.flags());        // default
    output(cout, ios_base::showpos);   // leading +
    output(cout, ios_base::boolalpha); // print true
    output(cout, ios_base::showpoint); // show .0

    output(cout, ios_base::boolalpha|ios_base::hex);  // bool on, base 16
    output(cout, ios_base::hex|ios_base::showbase);   // show 0x in front
    return 0;
}
```

formatparams.cpp

OUTPUT

```
prompt> formatparams
oldflags: 513   new: 513   12.47   1    99      255
oldflags: +513  new: +32   +12.47  1    +99     +255
oldflags: 513   new: 16384 12.47   true 99      255
oldflags: 513   new: 16    12.4700 1    99.0000 255
oldflags: 201   new: 4800  12.47   true 99      ff
oldflags: 0x201 new: 0x808 12.47   1    99      0xff
```

B.1.3 Stream Functions

We've used stream functions `fail` and `open` and mentioned `close` in Chapter 6. We summarize these and a few other stream functions here.

- `open(const char *)` opens an `ifstream` bound to the text file whose name is an argument. We use `string::c_str()` to obtain the c-string pointer needed as an argument. It's possible to `open` an output file for appending rather than writing, in general `open` takes an optional second argument we haven't used:
  - `ios_base::app`, open output for appending
  - `ios_base::out`, open a stream for output
  - `ios_base::in`, open a stream for input
  - `ios_base::binary`, open for binary i/o
  - `ios_base::trunc`, truncate to zero length

- `fail()` returns true if an i/o operation has failed, but characters have not been lost. You may be able to continue reading after calling `clear`.
- `clear()` clears the error state of the stream. After a stream has failed it must be cleared before i/o will succeed.
- `good()` returns true if a stream is in a good state. This is a nearly useless function, "good" isn't well defined. You shouldn't need to ever call `good`.
- `close()` flushes any pending output and manages all system resources associated with a stream. Many operating systems have a limit on the number of files that can be opened at one time. You don't often need to call `close` explicitly, it's called by the appropriate destructor.
- `eof()` returns true if the end-of-file condition of a stream is detected. This is another worthless function (see `good`). If `fail` is true, this function may be able to tell you if `fail` is true because end-of-file is reached.
- `ignore(int n, int sentinel)` skips as many as `n` characters, stops skipping when the `sentinel` character is read or when `n` characters are read, whichever comes first.
- `seekg(streampos p)` seeks an input stream to a position `p`. We use `seekg(0)` to reset a stream in the class `WordStreamIterator`; other uses are illustrated in the next section.

B.2 Random Access Files

We've used `ifstream` and `ofstream` streams as character-based streams, all the input and output is done a character at a time. Although operators `operator <<` and `operator >>` make it possible to insert and extract values of many types without reading one character at a time, underneath the streams are still character based.

Occasionally files are written as raw binary data rather than as character-based text. If you think you must write binary files, you may be correct, but you'll give up a great deal.

- Binary files aren't readable (as text) in a text editor, so you can't examine them without writing a program to help and you can't fix mistakes without writing a program.
- If you're writing objects whose size isn't fixed, such as strings, or objects containing pointers, you'll need to do lots of work to use files of raw binary data.

Seeking to a Fixed Position in a File. The file methods `seekg` and `tellg` shown in Program B.4 can be applied to text files as well as to binary files. Since text files are character based, seeking is based on the size of a character. Input files have a **get position**, which can be moved using `seekg` and whose position can be obtained using `tellg`, where the 'g' is for *get*. Similarly, output files use `seekp` and `tellp` for the **put position**. Using the seek and tell functions makes it possible to randomly access data in a file, as opposed to the sequential access we've used so far. Here random access means that it's possible to jump to location p without reading locations 0 through p-1, just as vectors have random access and linked-lists do not.

You'll need to consult a more advanced book on C++ for more information. A careful reading of *binaryfiles.cpp*, Program B.4, will show how to work with binary files. Program B.4 writes two files of `Date`s, one in text format, one as raw binary data. The low-level stream functions `read` and `write` manage a chunk of memory for reading or writing. The functions assume the memory is C-style array of characters; to interpret the memory as something else it must be cast to the appropriate type as shown by using the `reinterpret_cast` operator.

Program B.4 binaryfiles.cpp

```
#include <iostream>
#include <string>
#include <fstream>
using namespace std;
#include "prompt.h"
#include "date.h"

// illustrates reading/writing raw bits, binary files

int main()
{
    string filename = PromptString("file for storing Dates: ");
    int limit =       PromptRange("# of Dates ",10,10000);
    Date today;
    int start = today.Absolute();
    string text = filename + ".txt";
    string binary = filename + ".bin";
    cout << "testing program on " << today << endl;

    ofstream toutput(text.c_str());     // open text file
    int k;
    for(k=start; k < start+limit; k++)  // write text form of dates
    {   toutput << Date(k) << endl;
    }
    toutput.close();

    // open binary file, write raw dates
    ofstream boutput(binary.c_str(),ios_base::binary);
    for(k=start; k < start+limit; k++)
    {   Date d(k);
        boutput.write(reinterpret_cast<const char *>(&d),sizeof(d));
    }
    boutput.close();

    // open input file to read raw dates from
    ifstream input(binary.c_str(),ios_base::binary);
    input.seekg(0,ios_base::beg);     // to the beginning
    streampos startp= input.tellg();  // position of start
    input.seekg(0,ios_base::end);     // seek to end of stream
    streampos endp = input.tellg();   // position of end
    int size = endp−startp;            // number of entries

    cout << "size of file: "  << size << ", # dates = "
         << size/sizeof(Date) << endl;

    // read alldates in file, start at front
    input.seekg(0, ios_base::beg);
    for(k=0; k < size/sizeof(Date); k++)
    {   input.read(reinterpret_cast<char *>(&today),sizeof(today));
        cout << today << endl;
    }
    return 0;
}
```

binaryfiles.cpp

To show why you don't want to use binary files, the first three lines of `bindate.txt` follow.

```
May 27 1999
May 28 1999
May 29 1999
```

Here are the first few characters in `bindate.bin`

```
^[^@^@^@^E^@^@^@\317^G^@^@^\
```

OUTPUT

```
prompt> binaryfiles
file for storing Dates: bindate
# of Dates  between 10 and 10000: 10
testing program on May 27 1999
size of file: 120, # dates = 10
May 27 1999
May 28 1999
May 29 1999
May 30 1999
May 31 1999
June 1 1999
June 2 1999
June 3 1999
June 4 1999
June 5 1999
```

B.3 I/O Redirection

UNIX and MS-DOS/Windows machines provide a useful facility for permitting programs that read from the standard input stream, `cin`, to read from files. As we've seen, it's possible to use the class `ifstream` to do this. However, we often use programs written to read from the keyboard and use the stream `cin` to read from files instead. You can use **input redirection** to do this. When you run a program that reads from `cin`, the input can be specified to come from a text file using the symbol < and the name of the text file. Running Program 6.7, *countw.cpp*, as shown in the following, indicates how input redirection works.

OUTPUT

```
prompt> countw < melville.txt
number of words read = 14353
prompt> countw < hamlet.txt
number of words read = 31956
```

The less-than sign, <, causes the program on the left of the sign (in this case, *countw*) to take its `cin` input from the text file specified on the right of the < sign. The operating system that runs the program recognizes when the text file has "ended" and signals end of file to the program *countw*. This means that no special end-of-file character is stored in the files. Rather, end of file is a state detected by the system running the program.

It is possible to run the word-counting program on its own source code.

OUTPUT

```
prompt> countw < countw.cpp
number of words read = 54
```

Among the *words* of Program 6.7, *countw.cpp*, are `"main()"`, `"{"`, `"(cin"`, and `"endl;"`. You should examine the program to see if you can determine why these are considered words.