# Chapter 3: C++ I/O for the C Programmer

How to learn C++:

> **Try to put into practice what you already know, and in so doing you will in good time discover the hidden things which you now inquire about.**
> — Henry Van Dyke, American clergyman, educator, and author.

## 3.1 Familiar Things in a New Language

In C, `scanf()` and `printf()` are only defined for the built-in types and the programmer must supply a format field specifier for each variable read or printed. If the field specifier does not agree with the type of the variable, garbage will result.

In contrast, C++ supports a generic I/O facility called "C++ stream I/O". Input and output conversion are controlled by the declared type of the I/O variables: you write the same thing to output a double or a string as you write for an integer, but the results are different. This makes casual input and output easier. However, controlling field width, justification, and output precision is a pain in C++. The commands to do these jobs are wordy, non-intuitive, and cannot be combined on the same line with ordinary output commands. You can always use C formatted I/O in C++; you may prefer to do so if you want easy format control. Both systems can be, and often are, used in the same program. However, in this class, please use only C++ I/O.

This section is for people who know the stdio library in C and want convert their knowledge to C++ (or vice-versa). Several I/O tasks are listed; for each the C solution is given first, followed by the C++ solution. Code fragments are given to illustrate each item. Boring but accurate short programs that use the commands in context are also supplied.

## 3.2 Include Files

C: `#include <stdio.h>`
C++:

- `#include <iostream>` for interactive I/O.
- `#include <iomanip>` for format control.
- `#include <fstream>` for file I/O.
- `#include <sstream>` for strings that emulate streams.
- `using namespace std;` to bring the names of included library facilities into your working namespace.

## 3.3 Streams and Files

A stream is an object created by a program to allow it to access a file, socket, or some other source or destination for data.

**Predeclared streams:**

- C: `stdin, stdout, stderr`
- C++: `cin, cout, cerr, clog`
- The C++ streams `cin` and `cout` share buffers with the corresponding C streams. The streams `stderr` and `cerr` are unbuffered. The new stream, `clog` is used for logging transactions.

**Stream handling.** When a stream is opened, a data structure is created that contains the stream buffer, several status flags, and all the other information necessary to manage the stream.

- C:
```
typedef FILE* stream;                /* Or include tools.h. */
stream fin, fout;
fout = fopen( "myfile.out", "w" );   /* Open stream for writing. */
fin = fopen( "myfile.in", "r" );     /* Open stream for reading. */
if (fin == NULL)                     /* Test for unsuccessful open. */
if (feof( fin ))                     /* Test for end of file. */
```

- C++: Stream classes are built into C++; a `typedef` is not needed. There are several stream classes that form a class hierarchy whose root is the class `ios`. This class defines flags and functions that are common to all stream classes. Below that are the two classes whose names are used most often: `istream` for input and `ostream` for output. The predefined stream `cin` is an `istream`; `cout`, `cerr`, and `clog` are `ostreams`. These are all sequential streams–data is either read or written in strict sequential order. In contrast, the `iostreams` permit random-access input and output, such as a database would require. Below all three of these main stream classes are file-based streams and strings that emulate streams.

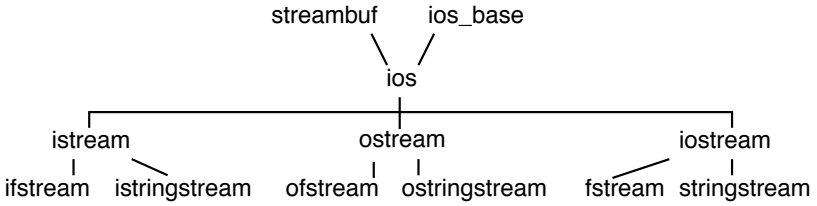

Figure 3.1: The stream type hierarchy.

Each class in the stream hierarchy is a variety of the class above it. When you call a function that is defined for a class high in the tree, you can use an argument of any class that is below that. For example, if `fin` is an `ifstream` (which is used for an input file) you can call any function defined for class `istream` or class `ios` with `fin` as an argument.

```
ofstream fout ( "myfile.out" );  // Open stream for writing.
ifstream fin ( "myfile.in" );    // Open stream for reading.
fin.open( "myfile.in" );         // Alternate way to open a stream.
fin.close();                     // Close a stream--not usually necessary.
if (!fin) fatal(...);            // Test for unsuccessful open.
if (fin.eof()) break;            // Test for end of file.
if (fin.good()) ...              // Test for successful read operation.
if (fin.fail()) ...              // Test for hardware or conversion error.
```

The stream declaration and open statement are combined in one line. The second declaration shown above is supposed to work, and works on my system. If the file does not exist, it is not created, and the `if` statement on the fifth line is used to handle the error. According to one student, though, this does not work properly in Visual C++ 4.1. If the file does not exist, this command creates it. To avoid creation of an empty file, he must open the file with the flag `ios::nocreate`, thus:

```
ifstream in ( "parts.in", ios::nocreate | ios::in );   // For Visual C++ ?
if (!in) fatal( "Could not open file parts.in" );
```

**Closing streams.** In both languages, streams are closed automatically when the program terminates. To close a stream prior to the end of the program:

- In C: `fclose( fin );`
- In C++: `fin.close();`

### 3.3.1 Common Stream Errors, Misconceptions, and Problems

**Stream ties:** `cout` is tied to `cin`, that is, whenever the cout buffer is non-empty and input is called for on `cin, cout` is automatically flushed. This is also true in C: `stdout` is tied to `stdin`. Both languages were designed this way so that you could display an input prompt without adding a newline to flush the buffer, and permit the input to be on the same line as the prompt.

**Flushing streams.** The standard `fflush()` function in C applies only to output streams. Some students have the belief that you can also flush a C input stream; this might be true in some nonstandard implementation, but it is not true according to the C standard.

In C++, `flush` is technically a manipulator, not a function, but it also applies only to `ostreams`. The `tools.cpp` library contains a definition of flush as a manipulator for `istreams`. This allows the programmer to leave the input buffer in a predictable state: empty. It is primarily useful for handling errors during interactive input.

```
// ------------------------------------------------------------
// Flush cin buffer as in cin >>x >>flush >>y; or cin >> flush;
istream&
flush( istream& is ) { return is.seekg( 0, ios::end ); }
```

**End-of-file and error processing.** In both C and C++, the end-of-file flag does not come on until you attempt to read data beyond the end of a file. The last line of data can be read fully and correctly without turning on the end-of-file flag. Therefore, an end-of-file test should be made between reading the data and attempting to process it. The cleanest, most reliable, simplest way to do this is by using an `if...break` statement to leave the input loop. The short program at the end of Section 3.6 shows how to test for hardware and format errors as well as eof.

## 3.4 Input

The following variables will be used throughout this section and the next:

```
char   c1;                  int    k1, k2;
short  n1, n2;              long   m1, m2;
float  x1, x2;              double y1, y2;
char*  word2 = "Polly";     char w1[80], w2[80], w3[80];
float* px1= &x1, *px2= &x2;
```

**Numeric input.** Basic operations are shown in the table below for both C and C++. Note how simple basic input is when using the C++ stream operators `>>` and `<<`. All of the operations in this table skip leading whitespace. It is not necessary and not helpful to do your own number conversion using `atoi` or `strtol`. Learn to use the stream library as it was intended!

The table below shows the normal way to read numbers. This works properly if the input is correct. However, it will cause problems if anything other than a number is in the input stream when a number is expected. If a non-numeric character is found while attempting a numeric read, the input stream will signal an error. A bullet-proof program tests the stream status flag using good() or fail() to detect such errors. To recover, use `clear()`. An example of read-error detection and proper error handling is shown at the bottom of the eof_demo program.

| Type | In C | In C++ |
|---|---|---|
| int | `scanf("%i%d", &k1, &k2);` | `cin >> k1 >> k2;` |
| long | `scanf("%li%ld", &m1, &m2);` | `cin >> m1 >> m2;` |
| short | `scanf("%hi%hd", &n1, &n2);` | `cin >> n1 >> n2;` |
| float | `scanf("%f%g ", &x1, &x2);` | `cin >> x1 >> x2;` |
| double | `scanf("%lf%lg", &y1, &y2);` | `cin >> y1 >> y2;` |

**Single character input.** Stream operator: `>>`. Function: `get(char_var)`.

| Operation | In C | In C++ |
|---|---|---|
| Read next keystroke | `scanf("%c", &c1);` | `cin.get(c1);` |
| Skip leading whitespace | `scanf(" %c", &c1);` | `cin >> c1;` |

**String input.** Functions: `ignore(n)`, `getline( buf, limit )`, `get( buf, limit, terminator )`, and `getline( buf, limit, terminator )`. Manipulator: `ws`.

| Operation | In C | In C++ |
|---|---|---|
| Skip leading whitespace, stop reading at next ws, no limit on string length: DO NOT DO THIS. | `scanf("%s", w1);` | `cin >> w1;` |
| Read up to first comma: | `scanf("%79[^,]", w1);` | `cin.get(w1, 80, ',');` |
| Read to and remove comma: | `scanf("%79[^,],", w1);` | `cin.getline(w1, 80, ',');` |
| Read line including `\n`: | `fgets(fin, w1, 79);` | `fin.getline(w1, 80);` |
| Read line excluding `\n`: | `scanf("%79[^\n]", w1);` | `cin.get(w1, 80);` |
| ... remove the newline | `(void)getchar();` | `cin.ignore(1);` |
| ... or | | `cin >> ws;` |
| Allocate space for string | `malloc(1+strlen(w1));` | |
| ... after `get()` | | `new char[1+fin.gcount()];` |
| ... after `getline()` | | `new char[fin.gcount()];` |

Note that the operations that use `>>` can be chained (combined in one statement) but those based on `get()` and `getline()` cannot. A single call on one of these functions is a complete statement

**Using `get()` and `getline()`.** Two input functions, `get()` and `getline()` are defined for class `istream` and all its derived classes. The difference is that `getline()` removes the delimiter from the stream and `get()` does not. Therefore, after using `get()` to read part of a line (up to a specified terminating character), you must remove that character from the input stream. The easiest way is to use `ignore(1)`.

After any read operation, the stream function named `gcount()` contains the number of characters actually read and removed from the stream. Saving this information (line 36 of the demonstration program at the end of this chapter) is useful when your program dynamically allocates storage for the input string, as in lines 40 and 41. The value returned by `gcount()` after using `getline()` will be one larger than the result after calling `get()` to read the same data.

**Whitespace.** When you are using `>>`, leading whitespace is automatically skipped. However, before reading anything with `get()` or `getline()`, whitespace must be skipped unless you *want* the whitespace in the input. Use `ws` for this purpose, not `ignore()`. This skips any whitespace that may (or may not) be in the input stream between the end of the previous input operation and the first visible keystroke on the current line. Usually, this is only one space, one tab, or one newline at the end of a prior input line. However, it could be more than one keystroke. By removing the invisible material using `ws` you are also able to remove any other invisible stuff that might be there.

## 3.5 Output

The C++ output stream, `cout`, was carefully defined to be compatible with the C stream, `stdout`: they write to the same output buffer. If you alternate calls on these two streams, the output will appear alternately on your screen. However, unless there is a very good reason, it is better style to stick to one set of output commands in any given program.

**Simple types, pointers, and strings.** The table shows alternative output format specifiers for several types. It uses the stream operator: `<<` and the stream manipulators `hex, dec`, and `endl`. Note that the string `"\n"`, the character `'\n'`, and the manipulator `endl` can all be used to end a line of output. For screen output they are equivalent. However, for file output there is one difference: the manipulator flushes the output stream; the character and string do not.

| Type | Lang. | Function call. |
|---|---|---|
| Numeric: | C | `printf("c1= %c k1= %i n1= %hd m1= %ld x1= %g y1= %g\n",`<br>`c1, k1, n1, m1, x1, y1);` |
| Numeric: | C++ | `cout <<"c1=" <<c1 <<" k1=" <<k1 <<" n1=" <<n1`<br>`<<" m1=" <<m1 <<" x1=" <<x1 <<" y1=" <<y1 <<"\n";` |
| `char[]` or `char*` | C | `printf("%s...%s %s\n", word, w2, w3);` |
| | C++ | `cout <<word <<"..." <<w2 <<" " <<w3 <<'\n';` |
| pointer in hexadecimal | C | `printf( "%p \n", px1 );` |
| | C++ | `cout <<px1 <<endl;` |
| `int`s in hexadecimal | C | `printf( "%x %x \n", k1, k2 );` |
| | C++ | `cout <<hex <<k1 <<' ' <<k2 <<dec <<endl;` |

**Output in hexadecimal notation.** Manipulators are used to change the setting of a C++ output stream. When created, all streams default to output in base 10, but this can be changed by "sending" the manipulator `hex` to the stream. Once the stream is put into hex mode it stays in hex until changed back by the `dec` manipulator.

**Field width, fill, and justification.** This table shows how to use the formatting functions `setw(), setfill()` and `setf()`. You must specify field width in C++ separately for every field. However, the justification setting and the fill character stay set until changed. Changing the justification requires a separate function call; `setf()` cannot be used as part of a series of `<<` operations.

| Style | How To Do It |
|---|---|
| In C, 12 columns, default justification (right). | `printf("%12i %12d\n", k1, k2);` |
| In C++, 12 cols, default justification (right). | `cout <<setw(12) <<k1 <<" " <<k2;` |
| In C, k1 in 12 cols, k2 in default width. | `printf("%12i %d\n", k1, k2);` |
| In C++, k1 12 cols (. fill), k2 default width | `cout <<setw(12) <<setfill('.')`<br>`<<k1 <<k2 <<endl;` |
| In C, two variables, 12 columns, left justified. | `printf("%-12i%-12d\n", k1, k2);` |
| In C++, twice, 12 columns, left justified. | `cout <<left <<setw(12) <<k1 <<setw(12) <<k2 <<endl;` |
| In C++, 12 columns, right justified, -fill. | `cout <<right <<setw(12) <<setfill('-') <<k1 <<endl;` |

**Floating point style and precision.** This table shows how to control precision and notation, which can be fixed point, exponential, or flexible . All of these settings remain until changed by a subsequent call.

| Style | | HowTo Do It |
|---|---|---|
| Default notation & precision (6) | C | `printf( "%g %g\n", y1, y2 );` |
| | C++ | `cout <<y1 <<' ' <<y2 <<endl;` |
| Change to precision=4 | C | `printf( "%.4g %.4g\n", y1, y2 );` |
| | C++ | `cout << setprecision(4) <<y1 <<' ' <<y2 <<endl;` |
| Fixed point, no decimal places | C | `printf( "%.0f \n", y1 );` |
| | C++ | `cout <<fixed <<setprecision(0) <<y1 <<endl;` |
| Scientific notation, default precision | C | `printf( "%e \n", y1 );` |
| | C++ | `cout <<scientific <<y1 <<endl;` |
| Scientific, 4 significant digits | C | `printf( "%.4e \n", y1 );` |
| | C++ | `cout <<scientific << setprecision(4) <<y1 <<endl;` |
| Return to default %g format | C++ | `cout.setf( 0, ios::floatfield );` |

**The old notation.** Older C++ compilers may not support the manipulators `fixed, scientific, right, and left`. If your compiler gives errors on these manipulators, you may need to use the older notation shown below.

| | |
|---|---|
| Right justification: | `fout.setf(ios::right, ios::adjustfield);` |
| Left justification: | `fout.setf(ios::left, ios::adjustfield);` |
| Fixed point notation. | `fout.setf(ios::fixed, ios::floatfield);` |
| Scientific notation. | `fout.setf(ios::scientific, ios::floatfield);` |

## 3.6 I/O and File Handling Demonstration

In this section, we give a program that uses many of the stream, input, and output facilities in C++. It should serve as a model for you in writing correct and robust file-handling programs. The main programis given first, followed by an input file , the corresponding output, and a header/code file pair that defines a structure named `Part` and its associated functions. Program notes are given after each file.

**The main program.**

```
1   #include "Part.hpp"
2
3   //---------------------------------------------------------------------
4   int main( void )
5   {
6       Part inventory[N];
7       ifstream instr( "parts.in" );
8       if ( !instr )  fatal( "Cannot open parts file %s", "parts.in" );
9
10      cerr << "\nReading inventory from parts input file.\n";
11      int n = get_parts( instr, inventory );                // Bad style!
12      instr.close();
13
14      cerr <<n <<" parts read successfully.\n\n";
15      print_parts( cout, inventory, n );
16      bye();
17  }
18
```

- Line 7 declares and opens the input stream; line 12 closes it. It is not necessary to close files explicitly; program termination will trigger closure. However, when you finish using a file during an early phase of program execution, it is a good practice to close it.

- This is a typical main program for C++: it delegates almost all activity to functions and provides output before every major step so that the user can monitor progress and diagnose malfunction.
- An integer variable is declared in the middle of the code on line 11. This is legal in C++, but is not really a good style to follow. This variable should probably be declared before line 7.
- Note that there is no return statement at the end of the main function. This is legal in C++ and follows the style used by some experts. Use the return statement at the end of `main` if your compiler gives warnings about omitting it. (Mine does not give an error comment.)

**Input.** In the input file, each data set should start with a part description, terminated by a comma and followed by two integers. The program should not depend on the number of spaces before or after each numeric field. The last line of the file should end in a newline character.

```
claw hammer,    57 3 9.99
claw hammer, 3 5  10.885
long nosed pliers, 57 15 2.34
roofing nails: 1 lb, 3 173 1.55
roofing nails: 1 lb, 57 85 1.59
```

**Output.** Using the given input file, the output looks like this:

```
Reading inventory from parts input file.
5 parts read successfully.

claw hammer.............57     3    9.99
claw hammer.............3      5   10.89
long nosed pliers.......57    15    2.34
roofing nails: 1 lb.....3    173    1.55
roofing nails: 1 lb.....57    85    1.59

Normal termination.
```

**The header file.** `part.hpp`.

```
24  /*  ------------------------------------------------------------------------
25  //  Header file for hardware store parts.                          Part.hpp
26  //  Created by Alice Fischer on Mon Dec 29 2003.
27  */
28  #include "tools.hpp"
29  #define BUFLENGTH 100   // Maximum length of the name of a part
30  #define N 1000          // Maximum number of parts in the inventory
31
32  struct Part{
33        char* part_name;
34        int store_code, quantity;
35        float price;
36  };
37
38  int get_parts( ifstream& fin, Part* data );
39  void print_parts( ostream& fout, Part* inventory, int n );
```

- The file `tools.hpp` includes all of the standard C and C++ header files that you are likely to need. If you include the tools source code file or the tools header file, you do not need to include the standard libraries. It also contains the line `using namespace std;`.
- Two constants are defined here because they are required by one of the two `Part` functions. The main program also uses `N`.
- In C++, you can use `struct` (line 32) without a typedef and without writing the keyword `struct` every time you use the type name. Note the syntax used in lines 6, 38, 39, etc.
- Note the ampersands in the second and third prototypes. They indicate that the stream parameters are passed by reference (not by value or by pointer). Most input and output functions take a stream parameter, and it is always a reference parameter.

**The implementation file, `part.cpp`.**

```
24  /*  ------------------------------------------------------------------------
25  //  Implementation file for hardware store parts.                 Part.cpp
26  //  Created by Alice Fischer on Mon Dec 29 2003.
27  */
28  #include "Part.hpp"
29  //-------------------------------------------------------------------------
30  int
31  get_parts( ifstream& fin, Part* data ) {
32      char buf[BUFLENGTH];
33      int len;            // Length of input string.
34      Part* p = data;     // Cursor to traverse data array.
35      Part* pend = p+N;   // Off-board pointer to end of array.
36
37      while (p<pend) {
38          fin >> ws;
39          if (fin.eof()) break;
40          fin.getline( buf, BUFLENGTH, ',' );
41          len = fin.gcount();
42          fin >> p->store_code >> p->quantity >> p->price;
43
44          if (fin.good()) {   // All required data items were read.
45              p->part_name = new char[len];
46              strcpy( p->part_name, buf );
47              ++p;            // Position cursor for next input.
48          }
49          else {
50              fin.clear();
51              cerr <<"Error reading line " <<(p-data+1) <<":\n"
52                   <<"    Before error: " <<buf <<" "
53                   <<p->store_code <<"  " <<p->quantity <<endl;
54              fin.getline( buf, BUFLENGTH ); // Skip rest of defective line.
55              cerr <<"    After error: " <<buf <<endl;
56          }
57      }
58      return p - data;  // Number of data lines read correctly and stored.
59  }
60
61  //-------------------------------------------------------------------
62  void
63  print_parts( ostream& fout, Part* inventory, int n ){
64      Part* p = inventory;
65      Part* pend = inventory+n;
66
67      for ( ; p<pend; ++p) {
68          fout <<left  <<setw(25) <<setfill('.') <<p->part_name;
69          fout         <<setw(3)  <<setfill(' ') <<p->store_code;
70          fout <<right <<setw(5)                 <<p->quantity;
71          fout <<fixed <<setw(8)  <<setprecision(2) <<p->price <<endl;
72      }
73  }
```

**Input: `get_parts`.** This function attempts to read the input file no matter what errors it may contain. If an error is discovered while reading a data set, the entire data set is skipped and the input stream is flushed up to the next newline character. The faulty input is reported.

- We use pointers to process the data array. In C++, as in C, pointers are simpler and more efficient to use than subscripts for sequential array processing. This function uses the normal pointer paradigm for an input function:
  - Line 34 sets a local cursor, `p`, to the head of the data array and line 35 sets a stationary pointer, `pend`, to the first address that is not part of the array.

  - The processing loop (lines 37...57) processes each array slot sequentially and increments the cursor (line 47) if good data is received.
  - Processing continues until the array becomes full (line 37, `while (p<pend)`) or until the end of the input file is reached. (The test is on line 39).

- Line 38 removes leading whitespace from the input stream. This is necessary before using `get()` or `getline()` in order to eliminate the newline character that terminated the previous line of input. If no whitespace is present in the stream, no harm is done. The I/O operator does remove leading whitespace, but we cannot use `>>` to read the part name because it stops reading at the first space and parts are often given names with multiple words.

- Line 39 breaks out of the input loop when end of file is encountered. This is the appropriate time to test for end of file because (a) we know there is no data, good or bad, to be processed and (b) the act of reading in all the whitespace will turn on the eof flag. This works whether or not there is a newline character on the last line of input, and whether or not the file is empty. DO NOT put your end-of-file test inside the `while` test on line 32.

- Line 40 reads characters from the input stream into the array named `buf`. Reading stops at the first comma, which is read but not stored in the input array. Instead, a null terminator is stored in the array. If a comma is found, `fin.good()` will be true. If no comma is found in the first `BUFLENGTH-1` characters, the read operation will stop and `fin.fail()` will be true.

- Line 41 sets `len` to the actual number of characters that were read and removed from the input stream. This number should be stored before doing any more input operations. After using `getline()`, it is the correct array length to use for allocating storage for the input string. (After calling `get()`, you need to add one to get the allocation length.) Lines 45 and 46 allocate storage, attach it to the `Part` array, and copy the input into it.

- Line 42 reads and stores three numbers. Whitespace before each number is automatically skipped.

- Line 44 tests whether all preceding input requests were successful. If so, we know that we have all four of the data fields needed to create a new part. Allocation was deferred until the input line was validated to avoid a problem known as a "memory leak". We want to be sure that all dynamically allocated storage is attached to the data array, so that it can be located and deallocated properly at a later time.

- Lines 49...56 handle errors. Control could come here because of a hardware error or because one of the data fields is missing. In either case, error recovery has two steps:

  - Line 45 calls `fin.clear()` to reset the error flags in the stream data structure. This must be done before normal operation can resume.
  - Line 49 clears the rest of the current line out of the input stream, in an effort to resynchronize the program and data. This code assumes that each data line ends in a newline character, so skipping to the next newline will position the file at the beginning of a data item.
  - In a real application, it can be important to know what data has been processed and what has been skipped because of errors. Lines 51...53 and 55 print all data that has not been processed on the `cerr` stream. This gives the user the information necessary to correct any errors.

- At any time, we can use pointer subtraction to compute the number of items that have been correctly stored in the array; simply subtract the pointer to the head of the array from the cursor. We return this value to the main program for use in all further processing of the data. There is no need to maintain a separate counter.

**Output: `print_parts`.** The input file was not formatted in neat columns. To make a neat table, we must be able to specify, for each column, a width that does not depend on the data printed in that column. Having done so, we must also specify whether the data will be printed at the left side or the right side of the column, and what padding character will be used to fill the space. In C, we can control field width and justification (but

not the fill character) by using an appropriate format. This lets us use a single `printf()` statement to print an entire line of the table (with the space character used as filler).

C++ I/O does not use formats, but we can accomplish the same goals with stream manipulators. However, we must use a series of operations that that control each aspect of the format independently. The output loop, below, illustrates the use of the formatting controls.

- The paradigm for an array output function is simpler than an array input function: we know that `n` data items must be printed, so the print loop is executed exactly `n` times. No validity checking or eof tests are necessary.
- To make the code layout easier to understand, this function formats and prints one output value per line of code.
- To print data in neat columns, the width of each column must be set individually. Lines 66, 67, and 70, and 71 use `setw()` to control the width of the four columns.
- Right justification is the default. This is appropriate for most numbers but is unusual for alphabetic information. We set the stream to left justification (line 68) before printing the part name. In this example, the first column of numbers (`store_code`) is also printed using left justification, simply to demonstrate what this looks like. The stream is changed back to right justification for the third and fourth columns.
- The default fill character is a space, but once this is changed, it stays changed until the fill character is reset to a space. For example, line 66 sets the fill character to `'.'`, so dot fill is used for the first column. Line 67 resets the filler to `' '`, so spaces are used to fill the other three columns.
- The price is a floating point number. Since the default formatting (`%g` with six digits of precision) is not acceptable, we must supply both the style (fixed point) and the precision (2 decimal places) that we need. Right justification and space as a fill character remain set from lines 70 and 69, but the field width must be supplied for every field.

**Bad files.** If the error and end-of-file tests are made in the right order, it does not matter whether the file ends in a newline character or not. (This will be covered in detail in the next section.) The output without a final newline is the same as was shown, above, for a well formed file. The result of running the program on an empty file is:

```
Reading inventory from parts input file.
0 parts read successfully.
```

**Bad data.** Suppose we introduce an error into the file by deleting the price on line three. The output becomes:

```
Reading inventory from parts input file.
Error reading line 3:
    Before error: long nosed pliers 57  15
    After error: roofing nails: 1 lb, 3 173 1.55
3 parts read successfully.

claw hammer.............57     3    9.99
claw hammer.............3      5   10.89
roofing nails: 1 lb.....57    85    1.59
```

The actual error was a missing float on the third line of input. The error is discovered when there is a type mismatch between the expectation (a number) and the `'r'` on the beginning of the fourth line. Because of the type mismatch, nothing is read in for the price, and the value in memory is set to 0. The faulty data is printed. Then the error recovery effort starts and wipes out the data on the line where the error was discovered. Details of the operation of the error flags are covered in the next section.

## 3.7 End of File and Error Handling

### 3.7.1 Using the command line.

Command-line arguments allow the programmer to select a data file at the time the command is given to run the program. This is especially useful when you wish to test the program with two different input files, as in the next program example. There are two ways to use file names as command line arguments:

1. Run the program from a command shell and type the file names after the name of the executable program.
2. In an integrated development environment (IDE), find a menu item that pops up a window that controls execution and enter the file names in the space provided. The location of these windows varies from one IDE to another, but the principle is always the same.

The program in this section illustrates how to use these arguments.

**In main(): Command-line arguments and file handling.**

1. Line 136 declares that the program expects to receive arguments from the operating system. If you have never seen command-line arguments, refer to Chapter 22 of *Applied C*, the text used in our C classes.
2. Line 140 lets us know that the user should supply two file names on the command line. Lines 142 and 144 pick up these arguments and use them to open two input streams. It is a good idea to open all files at the beginning of the program, especially if a human user will be entering data. There is nothing more frustrating that working for a while and *then* discovering that the program cannot proceed because of a file error.
3. Lines 143 and 145 test both streams to be sure they are properly open.

```
130  //-----------------------------------------------------------------------------
131  // C++ demonstration program for end of file and error handling.
132  // A. Fischer, April 2001                                        file: main.cpp
133  #include "tools.hpp"
134  #include "funcs.hpp"
135
136  int main( int argc, char* argv[] )
137  {
138      banner();
139      // Command line must give the name of the program followed by 2 file names.
140      if (argc < 3)  fatal( "Usage: eof_demo textfile numberfile" );
141
142      ifstream alpha_stream( argv[1] );
143      if (! alpha_stream)  fatal( "Cannot open text file %s", argv[1] );
144      ifstream num_stream( argv[2] );
145      if (! num_stream)  fatal( "Cannot open numeric file %s", argv[2] );
146
147      cout <<"\nIOstream flags are printed after each read in order:\n"
148           <<"goodbit : eofbit : failbit : badbit :\n";
149
150      use_get( alpha_stream );
151
152      alpha_stream.close();                   // Close file.
153      ifstream new_stream ( argv[1] );    // Re-open the file to use it again.
154      if (! new_stream)  fatal( "Cannot open text file %s", argv[1] );
155
156      use_getline( new_stream );
157      use_nums( num_stream );
158  }
```

4. Line 152 closes the input stream so that we can open it again and start all over using `getline()` instead of `get()`.
5. Line 153 reopens the stream using an additional parameter that seems to be necessary in Visual C++ but is optional according to the standard and in other C++ compilers that I use.

6. This program calls three functions that illustrate end-of-file and error handling with three different kinds of read operations. The numeric input file is shown in the next section with the function that processes it.
7. End-of-file handling interacts with error handling and with the way the data file ends. Two text files were used to test this program: one with and the other without a newline character on the end of the last line. The two text files and the output generated from each are shown after the code for the two functions.

### 3.7.2 Reading Lines of Text

It is possible to check for errors after reading from an input stream. If this is not done, and a read error or format incompatibility occurs, both the input stream and the input variable may be left containing garbage. After detecting an input error, you must clear the bad character or characters out of the stream and reset the stream's error flags. The program in this section shows how to detect read errors and handle end-of-file and other stream exception conditions.

**A function that uses `get()` to read lines of text.** There is only one difference in the operation of `get()` and `getline()`: the first does not remove the terminating character from the input stream, the second does. However, this small difference affects end-of-file and error handling. This pair of examples is provided to show the differences and to provide guidance about handling them.

```
159  //=============================================================================
160  // The get function leaves the trailing \n in the input stream.
161  // A. Fischer, April 2001                                        file: get.cpp
162  //-----------------------------------------------------------------------------
163  #include "tools.hpp"
164
165  void use_get( istream& instr )
166  {
167      cout <<"\nUsing get to read entire lines.\n";
168      char buf[80];
169      while (instr.good() ){
170          instr >> ws;                // Without this line, it is an infinite loop.
171          instr.get( buf, 80 );
172          cout <<instr.rdstate() <<" = ";
173          cout <<instr.good() <<":" <<instr.eof() <<":" <<instr.fail() <<":"
174               <<instr.bad() <<": ";
175          if (!instr.fail() && !instr.bad()) cout << buf << endl;
176      }
177      if ( instr.eof() ) cout << "---------------------------------\n" ;
178      else cout << "Low-level failure; stream corrupted.\n" ;
179  }
```

**Notes on the use_get() function.**

1. Line 169 demonstrates the use of an eof and error test as the `while` condition. The `good()` function tests for both eof and errors, so it is safer to use than the more common `while (!instr.eof())`.
2. Line 171 demonstrates the use of `get()`. The error flags are printed immediately after the read operation and before the line is echoed. The output shows that three read operations were performed, altogether. The third read went past the end of the file and caused the stream's eof flag to be set.
3. The good() function returns true (which is printed as 1) when none of the three exception flags are turned on. The `fail()` function returns a true result if no good data was processed on the prior operation. The `bad()` function returns true if a fatal low level IO error occurred, and the eof() function returns true when and end-or-file condition has occurred. These status bits remain set until explicitly cleared.

   As shown in the output, `eof()` can be true when good data was read, and also when no good data was read. (`fail()` is true).

4. On Line 172, we print a summary of the error conditions by writing `cout << instr.rdstate()`. The state value that is printed is the sum of the values of the stream flags that are set: eofbit=1, failbit=2, and badbit=4. Thus, when failbit and badbit are both turned on, the value of the state variable is 2 + 4 = 6.
5. Line 174 tests for the presence of good data. The test will be false when the read operation fails to bring in new data for any reason. We test it before printing or processing the data because we do not want to process old garbage from the input array. The fail flag was turned on after the fourth read operation because there was nothing left in the stream.
6. Using data or printing it without checking for input errors is taking a risk. A responsible programmer does not permit garbage input to cause garbage output. If `fail()` is true, the contents of the input variable are unreliable. If they are used after a failed get, the contents of the last correct input operation may be processed again. (This is a common problem.)
7. When using `get()` or `getline()`, the input variable may or may not be cleared to the empty string when `fail()` is true. My system does clear it, but I can find no mention of this in my reference books. It may be up to the compiler designer whether it is cleared or continues to hold the data from the last good read operation. Until this is clearly documented in reliable reference books, programmers should not depend on having the old garbage cleared out.

**A function that uses getline() to read lines of text.**

```
180  //================================================================================
181  // The getline function removes the trailing \n from the stream and discards it.
182  // A. Fischer, April 2001                                        file: getline.cpp
183  //--------------------------------------------------------------------------------
184  #include "tools.hpp"
185
186  void use_getline( istream& instr )
187  {
188      cout <<"\nUsing getline to read entire lines.\n";
189      char buf[80];
190      while (instr.good()) {
191          instr.getline( buf, 80 );
192          cout <<instr.rdstate() <<" = ";
193          cout << instr.good() <<":" << instr.eof() <<":" << instr.fail() <<":"
194               << instr.bad() <<": ";
195          if (!instr.fail()) cout << buf << endl;
196      }
197      cout << "-----------------------------------\n";
198  }
```

**Notes on the use_getline() function.**

1. Lines 186...198 demonstrate the use of `getline()`. Looking at the output, you can see that the third line *was* processed, whether or not it ended in a newline character. However, the outputs are not identical. The eofbit is turned on after the third line is read if it does not end in a newline, and after the fourth read if it does. This makes it very important to read, test for eof, and process the data in the right order.
2. When using `getline()`, good data and end-of-file can happen at the same time, as shown by the last line of output on the left. Therefore, we must make the test for good data first, process good data (if it exists) second, and make the eof test third. Combining the two tests in one statement, anywhere in the loop will cause errors under some conditions.
3. In this example, we read the input, print the flags, then test whether we *have* good input before printing it. In the `use_get()` function, we tested for both kinds of errors. A less-bullet-proof alternative would be to take a risk, and not test for low-level I/O system errors. Since these are rare, we usually do not have a problem when we omit this test. Code that implements this scheme would be:

```
while(instr.good()) {
    instr.getline( buf, 80 );
    if (! instr.fail()) { Process the new data here. }
}
```

**Input files for the EOF and error demo.**

**The file eofDemo.in:**

```
First line.
Second line.
Third line, without a newline on the end.
```

**The file eofDemo2.in:**

```
First line.
Second line.
Third line, with a newline on the end.
```

**Output from use_get().** Compare the results shown here on files without (left) and with (right) a newline character on the end of the file. In both cases, the data was read and processed correctly. This is only possible when the error indicators are checked in the "safe" order, that is, the `good()` function was called *before* checking for `eof()`. This allows us to capture the good data from the last line. If the outcome flags are checked in the wrong order, the contents of the last line may be lost. The IOstream flags are printed after each read in this order: goodbit : eofbit : failbit : badbit.

**Output from use_get() reading eofDemo.in.**

```
Using get to read entire lines.
0 = 1:0:0:0: First line.
0 = 1:0:0:0: Second line.
2 = 0:1:0:0: Third line, no newline on end.
-----------------------------------
```

**Output from use_get() with eofDemo2.in.**

```
Using get to read entire lines.
0 = 1:0:0:0: First line.
0 = 1:0:0:0: Second line.
0 = 1:0:0:0: Third line, with newline on end.
6 = 0:1:1:0: -----------------------------------
```

**Output from use_getline() using eofDemo.in.**

```
Using getline to read entire lines.
0 = 1:0:0:0: First line.
0 = 1:0:0:0: Second line.
2 = 0:1:0:0: Third line, no newline on end.
-----------------------------------
```

**Output from use_getline() with eofDemo2.in.**

```
Using getline to read entire lines.
0 = 1:0:0:0: First line.
0 = 1:0:0:0: Second line.
0 = 1:0:0:0: Third line, with newline on end.
6 = 0:1:1:0: -----------------------------------
```

### 3.7.3 EOF and error handling with numeric input.

Reading numeric input introduces the possibility of non-fatal errors and the need to know how to recover from them. Such errors occur during numeric input when the type of the variable receiving the data is incompatible with the nature of the input data. For example, if we attempt to read alphabetic characters into a numeric variable.

```
200  //================================================================================
201  // Handle conflicts between input data type and expected data type like this.
202  // A. Fischer, April 2001                                        file: getnum.cpp
203  //--------------------------------------------------------------------------------
204  #include "tools.hpp"
205
206  void use_nums( istream& instr )
207  {
208      cout <<"\nReading numbers.\n";
209      int number;
210      for(;;) {
211          instr >> number;
212          cout <<instr.good() <<":" <<instr.eof()<<":"<<instr.fail()<<":"<<instr.bad()<<": ";
213          if (instr.good()) cout << number << endl;
214          else if (instr.eof() ) break;
215          else if (instr.fail()) {        // Without these three lines
216             instr.clear();               // an alphabetic character in the input
217             instr.ignore(1);             // stream causes an infinite loop.
218          }
219          else if (instr.bad())           // Abort after an unrecoverable stream error.
220              fatal( "Bad error while reading input stream." );
221      }
222      cout << "-----------------------------------\n" ;
223  }
```

1. Lines 200...221 show how to deal with random numeric input errors. The file errDemo.in has three fully correct lines surrounding one line that has erroneous letters on it.

**The file errDemo.in:**

```
1
278
45abc
6
```

**Output from get_nums() using errDemo.in:**

```
Reading numbers.
1:0:0:0: 1
1:0:0:0: 278
1:0:0:0: 45
0:0:1:0: 0:0:1:0: 0:0:1:0: 1:0:0:0: 6
0:1:1:0: ----------------------------------
```

2. The first three numbers are read correctly, and the "abc" is left in the stream, unread, after the third read. When the program tries to read the fourth number, a conversion error occurs because 'a' is not a base-10 digit. The failbit is turned on but the eofbit is not, so control passes through line 214 to line 215, which detects the conversion error.
3. Lines 216 and 217 recover from this error. First, the stream's error flags are cleared, putting the stream back into the `good` state. Then a single character is read and discarded. Control goes back to line 210, the top of the read loop, and we try again to read the fourth number.
4. We cleared only one keystroke from the stream, leaving the "bc". So another read error occurs. In fact, we go through the error detection and recovery process three times, once for each non-numeric input character. On the fourth recovery attempt, a number is read, converted, and stored in the variable `number`, and the program goes on to normal completion. Compare the input file to the output: the "abc" just "disappeared", but you can see there were three "failed" attempts to read the 6.
5. Because of the complex logic required to detect and recover from format errors, we cannot use the eof test as part of a while loop. The only reasonable way to handle this logic is to use a blank `for(;;)` loop and write the required tests as `if` statements in the body of the loop. Note the use of the `if...break`; please learn to write loops this way.

## 3.8 Assorted short notes.

**Reading hexadecimal data.** The line `45abc` is a legitimate hexadecimal number, but not a correct base-10 number. Using the same program, we could read the same file correctly if we wrote line 211 as:

`instr >> hex >> number;` instead of `instr >> number;`

In this case, all input numbers would be interpreted as hexadecimal and the characters "abcdef" would be legitimate digits. The output shown below is still given in base-10 because we did not write `cout<<hex` on line 199. The results are:

```
Reading numbers.
1:0:0:0: 1
1:0:0:0: 632
1:0:0:0: 285372
1:0:0:0: 6
0:1:2:0: ----------------------------------
```

**Appending to a file.** A C++ output stream can be opened in the declaration. When this is done, the previous contents of that file are discarded. To open a stream in append mode, you must use the explicit `open` function, thus:

```
ofstream fout;
fout.open( "mycollection.out",  ios::out | ios::app );
```

The mode `ios::out` is the default for an ofstream and is used if no mode is specified explicitly. Here, we are specifying append mode, "app", so we also be write the "out" explicitly.

**Reading and writing binary files.** The `open` function can also be used with binary input and output files:

```
ifstream bin;
ofstream bout;
bin.open( "pixels.in", ios::in | ios::binary );
bout.open( "pixels.out",  ios::out | ios::binary );
```