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 can be a pain in C++. The commands to do these jobs are referred to as "manipulators" in C++ parlance and are defined in the `<iomanip>` library. They tend to be wordy, non-intuitive, and some cannot be combined with ordinary output commands on the same line. 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, such as a terminal window.

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 use and manage the stream.

Streams in C:

```
#include <stdio.h>;                    // Or include tools.hpp.
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 )                            // Test for unsuccessful open.
if (feof( fin ))                       // Test for end of file.
```

Streams in C++:

```
#include <stream>;              // Or include tools.hpp.
#include <fstream>;             // Or include tools.hpp.
#include <iomanip>;             // Or include tools.hpp.

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.
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.
```

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 `ios` 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` permits 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.

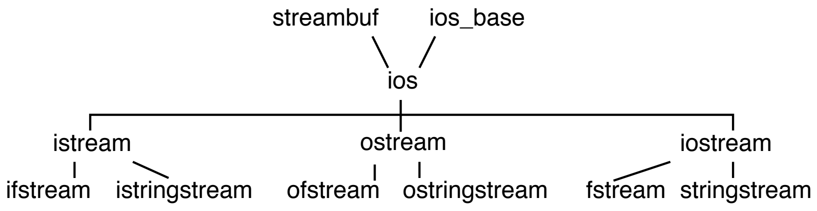

Figure 3.1: The stream type hierarchy.

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.

The stream declaration and open statement are combined in one line. If you attempt to open a file that does not exist, the operation fails and returns a null pointer, which must be tested.

In some older C++ implementations, this does not work properly. If the file does not exist, this command creates it. To avoid creation of an empty file, a programmer must open the file with the flag `ios::nocreate`, thus:

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

Buffered and unbuffered streams. When using `cout` or an `ofstream`, information goes into the stream buffer when an output statement is executed. It does not go out to the screen or the file until the buffer is *flushed*. In contrast, something written to `cerr` is not buffered and goes immediately to the screen.

When using `cin`, a line of text stays in the keyboard buffer until the return key is pressed. This allow the user to backspace and correct errors. When return is pressed, the entire line goes from the keyboard buffer to the stream buffer, where it stays until called for by the program. The program can read the entire line or any part of it. The rest will remain in the stream buffer, available for future read operations. In general, newlines in the input are treated the same as spaces, which allows a user to type several inputs on one line.

Flushing buffered output streams. The output from a buffered stream stays in the stream buffer until it is flushed. This happens under the following circumstances:

- When the buffer is full.
- When the stream is closed.
- When the program outputs an `endl` on any buffered output stream. `endl` is a manipulator that ends the output line, but its semantics are subtly different from a newline character.
- In C, when a newline is output to an interactive stream.
- In some C++ implementations, when a newline is output to an interactive stream. These implementations produce unnecessarily poor performance.
- For an output stream, when the program calls `flush` in C++ or `fflush` in C.

Stream ties. The stream `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 input streams. The built-in `flush` and `fflush` apply only to output streams. Some students are convinced that that `flush` also works on input stems. It does not, but they are confused because of the stream ties. However, after detecting an error in an interactive input stream, it is usually a good idea to flush away whatever remains in the input stream buffer. For thus purpose, the `tools.cpp` library contains a definition of flush as a manipulator for `istreams`. This allows the programmer to leave the input stream buffer in a predictable state: empty.

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

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

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

When should you close streams explicitly?

- Always, if you want to develop good, clean, habits that will never mislead you.
- Always, when you are done using a stream well before the end of execution.
- Always, when you are using Valgrind or a simlar tool to help you debug.

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 if there is are one or more newline characters on the end of that line. 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 3.9 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++. Most C++ I/O is done using two operators:

- For input, the *extraction* operator, `>>` extracts characters from an input stream and stores them in program variables.
- For output, the *insertion* operator, `<<` inserts values into an output stream.

Both of these C++ operators are polymorphic: they are predefined for all the built-in types and can be extended to handle any program-defined class. Note how simple basic input is when using these stream operators.

The table below shows the normal way to read numbers. All of these C and C++ methods for numeric input skip leading whitespace, read a number, and stop at the first whitespace or non-numeric input character. 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!

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;`

These methods work properly if the input is correct. However, if a number is expected, and anything other than a number is in the input stream the stream will signal an error. A bullet-proof program tests the stream status to detect such errors using the strategies explained in Section 3.7. The `clear()` function is used to recover from such errors.

String input. The simplest form for a string input command is something that should NEVER be used. It skips leading whitespace, reads characters starting with the first non-whitespace character, and stops reading at next whitespace. Ther is no limit on length of the string that is read. **DO NOT DO THIS!** If you try to read an indefinite-length input into a finite array, and you do not specify a length limit, the input can overflow the boundaries of the array and overlay nearby variables. A program that makes this error is open to exploitation by hackers.

In C: `scanf("%s", w1);`

In C++ `cin >> w1;`

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

Operation	In C	In C++
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.

Single character input. In C, the `"%c"` format is unlike all of the other formatted input methods because it does *not* skip leading whitespace. To skip the whitespace, the programmer must put a space in the format before the `%`. This leads to endless confusion and errors. In C++, this problem has been fixed, and using `>>` *always* skips leading whitespace . C and C++ both also provide a way to read raw characters. In C, this can be done with `getchar()` or `fgetc()`. In C++ it is done with `get()`.

Operation	In C	In C++
Skip leading whitespace, then read one character	`scanf(" %c", &c1);`	`cin >> c1;`

Read next keystroke	`scanf("%c", &c1);`
	`c1 = getchar();`
	`c1 = fgetc();`
		`cin.get(c1);`

This is useful when a program needs to know exactly what characters are in the input stream, and does not want whitespace skipped or modified. Example: a program that compresses a file.

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)`.

The `gcount()` 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 explicitly skipped unless you *want* the whitespace in the input. Use the `ws` manipulator 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. For us, this means that you should use C++ style consistently.

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.

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", c1, k1, n1, m1, x1, y1);`
Numeric:	C++	`cout <<"c1=" <<c1 <<" k1=" <<k1 <<" n1=" <<n1 <<" m1=" <<m1 <<" x1=" <<x1 <<" y1=" <<y1 <<"\n";`
`char[]` or	C	`printf("%s...%s %s\n", word, w2, w3);`
`char*`	C++	`cout <<word <<"..." <<w2 <<" " <<w3 <<'\n';`
pointer in	C	`printf( "%p \n", px1 );`
hexadecimal	C++	`cout <<px1 <<endl;`
`ints` in	C	`printf( "%x %x \n", k1, k2 );`
hexadecimal	C++	`cout <<hex <<k1 <<' ' <<k2 <<dec <<endl;`

Manipulators A *manipulator* in C++ is a function whose only parameter is an input stream or an output stream. It modifies the stream and returns that same stream. The flush function (shown above and defined in tools.hpp) is a manipulator. This is an important kind of function because stream manipulators can be written in the same chain of input or output commands as the data. This makes it much easier to write well-formatted output.

The C++ standard defines many useful manipulators but misses some that are useful, such as `flush()` for an input stream. Another useful manipulator is shown below. The language standard provides manipulators to change numeric output from the default format (like `%g` in C) to fixed point `%f` or scientific `%e` notation. However it does not provide a manipulator to change back to the default `%g` format. This function does the job:

```
ostream& general( ostream& os ){ // Use:  cout <<fixed <<x <<general <<y;
    os.unsetf( ios::floatfield );
    return os;
}
```

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 writing the manipulator `hex` in the output chain. Once the stream is put into hex mode it stays in hex until changed back by the `dec` or `oct` manipulator.

Field width, fill, and justification. This table shows how to use the formatting functions `setw()`, `setfill()` and `setf()`. Note: 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('.') <<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 <<general;` // Defined in tools.

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, that manipulates the fields inside the stream object.

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 Generic Insertion Sort

The purpose of this section is to illustrate interactive input and output in the context of a simple one-class program.. The code embodies a variety of OO design, coding, and style principles, and also a few old C techniques. Notes follow the code. Use this example as a general guide for doing your own work. The most important themes are:

- Highly modular code with a streamlined main function.
- A class with all the normal parts: data, constructor, destructor, and public functions. The MemList class is an array packaged together with the information needed to use and manage it: the number of slots in the array and the number of slots that are currently filled with valid data.
- Encapsulation: this class takes care of itself. The functions that belong to the MemList class are the only ones that operate on the class's data members.

A type declaration and the main program are given first, followed by detailed notes, keyed to the line numbers in the code. A call chart for the program is given in Figure 2.1. In the chart, white boxes surround functions defined in this application, light gray boxes denote functions in the standard libraries and dark gray denotes the tools library.

3.6.1 Main Program

```
 1  // ---------------------------------------------------------------------------
 2  //  Main program for the Membership List.                              main.cpp
 3  //  Created by Alice Fischer on Aug 23 2014.
 4  // ---------------------------------------------------------------------------
 5  //  This program reads a set of names from the keyboard, stores and sorts them,
 6  //  and displays the sorted list.
 7  // ---------------------------------------------------------------------------
 8  #include "tools.hpp"
 9  #include "memlist.hpp"
10
11  // ----------------------------------------------------------------------------
12  int main( void )
13  {
14      banner();
15      MemList theData;
16      cout << "Constructed empty MemList" ;
17
18      cout << "\nEnter names of your club members, period to finish. \n" ;
19      theData.readData( cin );
20
21      cout <<"Beginning to sort.\n";
22      theData.sortData();
23
24      cout <<"Sorted results:\n";
25      theData.printData( cout );
26      bye();
27      return 0;
28  }
```

1. A program that is built in modules has a main program plus a pair of files for each module: a header file and a code file. The code file starts with a command to include the corresponding header file. The main program starts with commands to include one or more module headers. In this case, `main()` uses the MemList module and the tools module, so we include the headers for those modules.
2. The main program instantiates the MemList object, named "theData".
3. The rest of `main()` (lines 22...27) is simply an outline of the processing steps with some user feedback added. All work is delegated to functions. A call chart (Figure 2.1) shows the overall dynamic structure of a program and is an important form of documentation. The arrows indicate which functions are called by each other function. Frequently, the standard I/O functions are omitted from such charts.

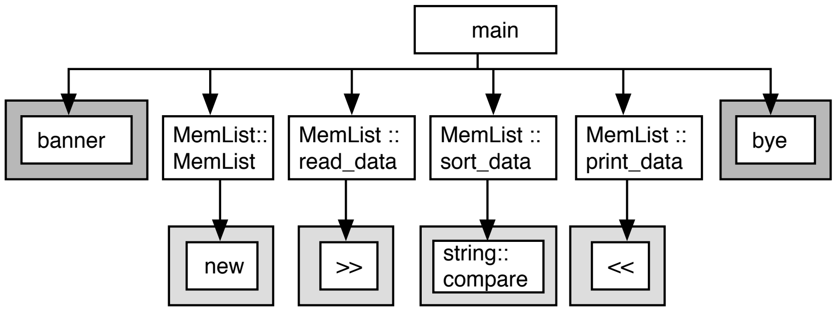

Figure 3.2: Call hierarchy chart for insertion sort.

3.6.2 The MemList Header File

```
29  // ---------------------------------------------------------------------------
30  //  Header file for the MemList program.                            memlist.hpp
31  //  Created by Alice Fischer on Aug 23 2014.
32  // ---------------------------------------------------------------------------
33  #include "tools.hpp"
34  #include <string>
35
36  class MemList {
37  private:
38      int n=0;            // Data items currently stored in the pack.
39      int max =100;       // Allocation length of the array.
40      string* store = new string[max];      // Dynamic data array.
41
42  public:
43      MemList() { }                          //Constructor
44      ~MemList() { delete store; }           // Destructor
45      // ------------------------------------------------------------- Prototypes
46      void readData( istream& instream );
47      void sortData();
48      void printData( ostream& outstream );
49  } ;
```

1. Type declarations, `#define`s, and prototypes are normally found in header files.
2. Line 33 includes definitions from the local library, `tools`, that we will be using throughout the term. Note the use of quotes, not angle brackets for a local library. Note also that we include the header file, not the code file, because we are doing a multi-module compile with a separate link operation.
3. Line 34 includes header for the standard C++ string library. We use angle brackets for standard library headers, and write only the library name, omitting the .h or .hpp.
4. This header is included here only for emphasis. We do not need to include it because we are including the tools, which includes `<string>`.
5. Lines 36–49 define the class MemList, which has three data members, a constructor, a destructor, and three functions.
6. The `MemList` type makes a coherent grouping of *all* the parts that are needed to manage an array of data. In C++. The data members are declared to be private. The member functions provide access to the data structure and they are the *only* way to access it.
7. C++ 11 and newer versions permit simple constant initializations of the data members of a class. Here we initialize `n`, the actual number of names currently stored in the array, and `max`, the maximum number of names that can be stored there.
8. The class constructor initializes the third data member by allocating dynamic storage for 100 strings. This definition is so short it fits on one line and is therefore fully defined in the header file. It will be executed automatically when a MemList object is declared or `newed`.
9. Corresponding to the `new` in the constructor is a `delete` in the destructor. This will free the dynamic memory when the MemList object is deallocated.
10. Three functions are given only prototypes here and defined fully in the matching .cpp file. They are too long to define within the class declaration.

3.6.3 The MemList Functions

1. The code file for a module must include its corresponding header file. Normally, it should not `#include` any other header files. The following paragraphs look at the implementations of the MemList functions.
2. The `readData()` function (lines 57. . . 66) reads whitespace-delimited names from the stream and uses them to fill he array with data. We call this from line 19 of main, with `cin` as the parameter. Therefore,

we will be reading input from the keyboard. Look at the output at the end of this section and note that it does not matter whether the user types a space or a newline. They are treated the same way by `>>`.

3. The last action in `readData()` is to store the amount of data received into the MemList's variable. This ensures that any function that processes the data can find out how much data is there. At no time is reference made to global variables or constants; the MemList is self sufficient.
4. The `printData()` function (lines 70. . . 76) prints an output header followed by the names in the list, one per line. We call it from line 22 of `main`, with `cout` as a parameter. Therefore we will display the output to the screen.

```
50  // ----------------------------------------------------------------------------
51  //  Code file for the MemList program.                              memlist.cpp
52  //  Created by Alice Fischer on Aug 23 2014.
53  #include "memlist.hpp"
54
55  //-----------------------------------------------------------------------------
56  // Use sequential access pattern to store data from infile into data pack.
57  void
58  MemList::readData( istream& infile )
59  {
60      int k;
61      for( k=0; k<max; ++k ) {
62          infile >> store[k];
63          if( store[k][0]=='.' ) break;   // Leave loop if input was a period.
64      }
65      n = k;                              // Actual # of items read and stored.
66  }
67
68  // ----------------------------------------------------------------------------
69  // Print array values, one per line, to the selected stream.
70  void
71  MemList::printData( ostream& outstream )
72  {
73      outstream << n <<" names were entered.\n";
74      for( int k=0; k < n; ++k)
75          outstream << store[k] <<endl;
76  }
77
78  // ----------------------------------------------------------------------------
79  // Generic insertion sort using a MemList.
80  // Sort n values starting at pData->store by an insertion sort algorithm.
81
82  void
83  MemList::sortData()
84  {
85      int pass;           // Subscript of first unsorted item; begin pass here.
86      int hole;           // Array slot containing no data.
87      string newcomer;    // Data value formerly in hole, now being inserted.
88
89      for ( pass = 1; pass<n; ++pass ) {
90          // Pick up next item and insert into sorted portion of array.
91          newcomer = store[pass];
92          for ( hole=pass; hole>0; --hole ) {
93              int moveit = hole-1;        // Subscript of slot next to the hole.
94              if ( newcomer.compare(store[moveit]) > 0 ) break; // Right slot found.
95              store[hole] = store[moveit];            // Move item up one slot.
96          }
97          store[hole] = newcomer;
98      }
99  }
```

Sorting by insertion. The `sortData()` function (lines 82. . . 99) implements insertion sort, the most efficient of the simple sorts. In practice, it is reasonable to use insertion sort for small arrays. This is a typical double-loop implementation of the insertion sort algorithm.

1. Each declaration has a comment that explains the usage of the variable. Note the use of brief, imaginative, meaningful variable names. This makes a huge difference in the readability of the code. In contrast, code is much harder to understand when it is written using identifiers like `i` and `j` for loop indices.
2. At all times, the array is divided into two portions: a sorted portion (smaller subscripts) and an unsorted portion (larger subscripts). Initially, the sorted part of the array contains one element, so the sorting loop (lines 89. . . 98) starts with the element at store[1].
3. The algorithm makes $N - 1$ passes over the data to sort N items. On each pass through the outer loop, it selects and copies one item (the `newcomer`, line 91) from the beginning of the unsorted portion of the array, leaving a `hole` that does not contain significant data.
4. The inner loop (lines 92. . . 96) searches the sorted portion of the array for the correct slot for the newcomer. The `for` loop is used to prevent overrunning the boundaries of the array. Within the loop body, an `if...break`, line 94, is used to leave the loop when the correct insertion slot is located.
5. To find that position, the loop scans backwards through the array, comparing the `newcomer` to each element in turn (line 94). If the `newcomer` is smaller, the other element is moved into the `hole` (line 95) and the `hole` index is decremented by the loop (line 92).
6. If the `newcomer` is not smaller, we break out of the inner loop and copy the `newcomer` into the `hole` (line 97).

The output.

```
100   ---------------------------------------------------------------
101   Alice Fischer
102   CS 620
103   Sat Aug 23 2014    16:25:06
104   ---------------------------------------------------------------
105   Constructed empty MemList
106   Enter names of your club members, period to finish.
107   ann vic
108   sammy
109   john
110   .
111   Beginning to sort.
112   Sorted results:
113   4 names were entered.
114   ann
115   john
116   sammy
117   vic
118
119   Normal termination.
```

3.7 End of File and Error Handling

3.7.1 Using the command line.

Command-line arguments allow the programmer to specify runtime parameters, such as a file name, at the time when the command is given to run a 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 command line arguments.

```
 1  //-----------------------------------------------------------------------------
 2  // C++ demonstration program for end of file and error handling.
 3  // A. Fischer, April 2001                                           file: main.cpp
 4  #include "tools.hpp"
 5  #include "funcs.hpp"
 6
 7  int main( int argc, char* argv[] )
 8  {
 9      banner();
10      // Command line must give the name of the program followed by 2 file names.
11      if (argc < 3)  fatal( "Usage: eofDemo textfile numberfile" );
12
13      ifstream alpha_stream( argv[1] );
14      if (! alpha_stream)  fatal( "Cannot open text file %s", argv[1] );
15      ifstream num_stream( argv[2] );
16      if (! num_stream)  fatal( "Cannot open numeric file %s", argv[2] );
17
18      cout <<"\nIOstream flags are printed after each read in order:\n"
19           <<"good : failb : eof: bad :\n";
20
21      use_get( alpha_stream );
22
23      alpha_stream.close();                   // Close file.
24      ifstream new_stream ( argv[1] );    // Re-open the file to use it again.
25      if (! new_stream)  fatal( "Cannot open text file %s", argv[1] );
26
27      use_getline( new_stream );
28      use_getnumhex( num_stream );
29  }
```

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

1. Line 7 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, or *www.cprogramming.com/tutorial/c/lesson14.html* "Accepting command line arguments in C using argc and argv".
2. Line 11 lets us know that the user should supply two file names on the command line. Lines 13 and 15 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 14 and 16 test both streams to be sure they are properly open.
4. Line 23 closes the input stream so that we can open it again and start all over using `getline()` instead of `get()`.
5. Line 24 and 25 reopen the file.
6. This program calls three functions that illustrate end-of-file and error handling with three different kinds of read operations.

3.7.2 Reading 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. A pair of examples is provided here to show the differences and to provide guidance about handling them.

Checking for read errors. It is important 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.

End-of-file handling interacts with error handling, with the way the line is read (`get()` or `getline()`) and with the way the data file ends (with or without a newline character). 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 are given first, then the code for the two read functions and the output generated from each.

Reading the stream status reports.

1. Line 42 calls `get()`. The stream status is printed immediately after the read operation and before the line is echoed, in this order: rdstate = good : fail : eof : bad :
2. Three flags (not four) are used to record the status of a stream. They are set by the system when any sort of an exception happens during reading, and they remain set until explicitly cleared.
    - The `fail()` function returns a true result if no good data was processed on the prior operation.
    - The eof() function returns true when an end-or-file condition has occurred. There may or may not be good data, also.
    - The `bad()` function returns true if a fatal low level IO error occurred.
3. There is no "good" bit. The good() function returns true (which is printed as 1) when none of the three exception flags are turned on. As shown in the output, `eof()` can be true when good data was read, and also when no good data was read. When eof happens and there is no good data, the `fail()` is true).
4. The value returned by `rdstate()` is a number between 0 and 7, formed by the concatenation of the status bits: fail/eof/bad.

Using `get()` to read lines of text.

```
30  //==============================================================================
31  // The get function leaves the trailing \n in the input stream.
32  // A. Fischer, April 2001                                           file: get.cpp
33  //------------------------------------------------------------------------------
34  #include "tools.hpp"
35
36  void use_get( istream& instr )
37  {
38      cout <<"\nUsing get to read entire lines.\n";
39      char buf[80];
40      while (instr.good() ){
41          instr >> ws;                // Without this line, it is an infinite loop.
42          instr.get( buf, 80 );
43          cout <<instr.rdstate() <<" = ";
44          cout <<instr.good()  <<":" <<instr.fail() <<":" <<instr.eof()
45               <<":" <<instr.bad() <<": ";
46          if (!instr.fail() && !instr.bad()) cout << buf << endl;
47      }
48      if ( instr.eof() ) cout << "-----------------------------------\n" ;
49      else cout << "Low-level failure; stream corrupted.\n" ;
50  }
```

Input files for the EOF and error demo.

The file eofDemo.in:

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

The file eofDemo2.in:

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

The output from `use_get()` with a normal file that ends in newline.

```
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 a newline on the end.
6 = 0:1:1:0: ----------------------------------
```

The output from `use_get()` with an abnormal file that has no final newline.

```
Using get to read entire lines.
0 = 1:0:0:0: First line.
0 = 1:0:0:0: Second line.
2 = 0:0:1:0: Third line, without a newline on the end.
----------------------------------
```

Notes on the `use_get()` function.
Compare the results shown here on files with (upper output) and without (lower output) 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, we check for good data *before* checking for `eof()`. This allows us to capture the good data from the last line in eofDemo2.in. If the outcome flags are checked in the wrong order, the contents of the last line will be lost.

1. The call on `get()` is inside a while loop. 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.
2. Line 40 demonstrates the use of an eof and error test as the `while` condition. The `good()` function tests whether good input has been received. It will return false if an error occurred, or if eof occurred before good data was read. Thus, it is safer to use than the more common `while (!instr.eof())`.
3. On Line 44, 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: fail=4, eof=2, and bad=1. Thus, when failbit and eofbit are both turned on, the value of the state variable is $4 + 2 = 6$.
4. Line 46 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.
5. 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 do not represent new data.
6. 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.

   If a program uses the input variable after a failed get, the contents of the last correct input operation may still be in the variable and might be processed a second time. (This is a common program error.)

Using getline() to read lines of text.

```
//===========================================================================
// The getline function removes the trailing \n from the stream and discards it.
// A. Fischer, April 2001                                      file: getline.cpp
//---------------------------------------------------------------------------
#include "tools.hpp"

void use_getline( istream& instr )
{
```

```
69      cout <<"\nUsing getline to read entire lines.\n";
70      char buf[80];
71      while (instr.good()) {
72          instr.getline( buf, 80 );
73          cout <<instr.rdstate() <<" = ";
74          cout <<instr.good()  <<":" <<instr.fail() <<":" <<instr.eof()
75               <<":" <<instr.bad() <<": ";
76          if (!instr.fail() && !instr.bad()) cout << buf << endl;
77      }
78      cout << "-----------------------------------\n";
79  }
```

Output from use_getline() on a normal file that ends in newline.

```
80  Using getline to read entire lines.
81  0 = 1:0:0:0: First line.
82  0 = 1:0:0:0: Second line.
83  0 = 1:0:0:0: Third line, with a newline on the end.
84  6 = 0:1:1:0: -----------------------------------
```

Output from use_getline() on an annormal file that lacks a final newline.

```
85  Using getline to read entire lines.
86  0 = 1:0:0:0: First line.
87  0 = 1:0:0:0: Second line.
88  2 = 0:0:1:0: Third line, without a newline on the end.
89  -----------------------------------
```

Notes on the use_getline() function.

1. Lines 61...79 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. As 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. }
}
```

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.

The `use_getnum()` function.

```
 90  // ==========================================================================
 91  // Handle conflicts between input data type and expected data type like this.
 92  // A. Fischer, April 2001                                    file: getnum.cpp
 93  //---------------------------------------------------------------------------
 94  #include "tools.hpp"
 95
 96  void use_getnum( istream& instr )
 97  {
 98      cout <<"\nReading numbers.\n";
 99      int number;
100      for(;;) {
101          instr >> number;
102          cout <<instr.rdstate() <<" = ";
103          cout <<instr.good()  <<":" <<instr.fail() <<":" <<instr.eof()
104               <<":" <<instr.bad() <<": ";
105          if (instr.good()) cout << number << endl;
106          else if (instr.eof() ) break;
107          else if (instr.fail()) {       // Without these three lines
108             instr.clear();              // an alphabetic character in the input
109             instr.ignore(1);            // stream causes an infinite loop.
110          }
111          else if (instr.bad())          // Abort after an unrecoverable stream error.
112              fatal( "Bad error while reading input stream." );
113      }
114      cout << "----------------------------------\n" ;
115  }
```

The file eofNumeric.in.

```
1
278
45abc
6
```

Output from use_getnum() with eofNumeric.in.

```
Reading numbers.
0 = 1:0:0:0: 1
0 = 1:0:0:0: 278
0 = 1:0:0:0: 45
4 = 0:1:0:0: 4 = 0:1:0:0: 4 = 0:1:0:0: 0 = 1:0:0:0: 6
6 = 0:1:1:0: ----------------------------------
```

1. Lines 105. . . 113 show how to deal with numeric input errors. The file eofNumeric.in has three fully correct lines surrounding one line that has erroneous letters on it.
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 105 to line 107, which detects the conversion error.
3. Lines 108 and 109 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 100, the top of the read loop, and we try again to read the fourth number.
4. After the first error, 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 101 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.
0 = 1:0:0:0: 1
0 = 1:0:0:0: 632
0 = 1:0:0:0: 285372
0 = 1:0:0:0: 6
6 = 0:1:1:0: ----------------------------------
```

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", but 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 );
```

3.9 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 is a sophisticated program that may be too much to handle initially. But at the point you need to know how to do bullet-proof file handling, come back to this example, and it will serve as a model for you in writing correct and robust file-handling programs.

The main program is 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.

Inventory: The main program.

- Line 13 declares and opens the input stream; line 18 closes it. It is not necessary to close files explicitly; program termination will trigger closure. However, it is good practice to close them.
- 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.
- The constant `N` is defined here because it is required by `readParts` and by the main program.
- An integer variable is declared in the middle of the code on line 17. This is legal in C++.
- The call on `bye()` on line 22 prints a termination message. As you begin to write more and more complex programs, calling `bye()`will become very useful for debugging.
- The tasks of reading and printing the data for a single part are delegated to the Part class. The main program contains the functions that read and print many parts.

```cpp
// Demo program for basic input and output. ----------------------------------
// Michael and Alice Fischer,  August 11, 2014
#define N 1000          // Maximum number of parts in the inventory
#include "Part.hpp"

int readParts( istream& fin, Part* data );
void printAll( ostream& fout, Part* inventory, int n );

//----------------------------------------------------------------------------
int main( void )
{
    Part inventory[N];
    ifstream instr( "parts.in" );
    if ( !instr )  fatal( "Cannot open parts file %s", "parts.in" );

    cerr << "\nReading inventory from parts input file.\n";
    const int n = readParts( instr, inventory );
    instr.close();

    cerr <<n <<" parts read successfully.\n\n";
    printAll( cout, inventory, n );
    bye();
    return 0;
}

//----------------------------------------------------------------------------
int readParts( istream& fin, Part* data ) {
    Part* p = data;     // Cursor to traverse data array.
    Part* pend = p+N;   // Off-board pointer to end of array.

    while (p<pend) {
        fin >> ws;
        bool gotData = p->read( fin );
        // First handle the normal case
        if (gotData) {
            ++p; // Position cursor for next input.
            if (fin.eof()) break;  // no newline at EOF
        }
        // No good data on current line
        // EOF with gcount()==0 means normal EOF with newline
        else if (fin.eof() && fin.gcount()==0) break;
        // Partial line or data conversion error
        else {
            cerr <<"Error reading line " <<(p-data+1) <<"\n";
            fin.clear();
            fin >>flush; // Skip rest of defective line.
        }
    }
    return p - data;  // Number of data lines read correctly and stored.
}

//----------------------------------------------------------------------------
void
printAll( ostream& fout, Part* inventory, int n ){
    Part* p = inventory;
    Part* pend = inventory+n;

    for ( ; p<pend; ++p) {
        p->print( fout );
        fout<<endl;
    }
}
```

Reading the file into the inventory array. [Lines 27 . . . 50]

- 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:
- Lines 28 and 29 set pointers to the beginning and end of the inventory array. The end pointer points to the slot past the last *allocated* array slot. There may or may not be enough data in the file to fill this array.
- The scanning pointer is named `p`. It is incremented on line 36, only if the program received good data.
- Line 49 uses pointer arithmetic to calculate the number of actual data items that have been stored in the array. The result of the subtraction is the number of array slots between the two pointers.
- 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).
- Error testing is done in the right order to catch all problems and properly process all data, whether or not there is a newline on the end of the file. This logic seems complex, but is probably the shortest and easiest way to do the job properly.
- Line 32 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 33 delegates the task of reading one line of data to the Part class, which is the expert on what constitutes a legal part description. `Part::read()` returns an error report: true if it found good data, false otherwise.
- Line 35 tests for good data. If it happened, the scanning pointer is incremented. Then the eof test is made. This is the test that ends the loop when there is no newline on the end of the file.
- Line 41 is the normal eof test that ends processing for files that end properly with a newline.
- Lines 43. . . 47 handle errors by clearing the stream's status flags and removing the rest of the defective input line from the stream.
- DO NOT put your end-of-file test inside the `while` test on line 31.

Printing the inventory array. [Lines 53. . . 62]

- Lines 55 and 56 set pointers to the beginning and end of the inventory array. The end pointer points to the slot past the last *actual* filled array slot.
- The for loop prints all data in the array by delegating printing to the Parts class.

The input file. 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. This is the input file used for testing:

```
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
```

The screen 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 for the Part module: `part.hpp`**.**

```
63  //-----------------------------------------------------------------------------
64  //  Header file for hardware store parts.                          Part.hpp
65  //  Created by Alice Fischer on Mon Dec 29 2003.
66  //
67  #include "tools.hpp"
68  #define BUFLENGTH 100   // Maximum length of the name of a part
69
70  //-----------------------------------------------------------------------------
71  class Part{
72  private:
73      char* partName;
74      int storeCode, quantity;
75      float price;
76  public:
77      bool read( istream& fin );
78      void print( ostream& fout );
79  };
```

- 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;`.
- In C++, you can use `struct` or `class` (line 26) to define a new type name. There is no need for a typedef and no need to write the keyword `struct` or `class` every time you use the type name. Note the syntax used in lines 6, 32, 33, etc.
- The data members of a class are the parts that you would declare for a —pl C `struct`. They are virtually always made private.
- In C++, the functions that operate on a structure are declared as part of the structure or class. Most, but not all function members are declared public so that other modules can call them.
- 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`**.**

```
80  //-----------------------------------------------------------------------------
81  //  Implementation file for hardware store parts.                  Part.cpp
82  //  Created by Alice Fischer on Mon Dec 29 2003.
83  //
84  #include "Part.hpp"
85  //-----------------------------------------------------------------------------
86  // Precondition: fin is open for input; no initial whitespace;
87  // eof flag is off
88  bool
89  Part::read( istream& fin ) {
90      char buf[BUFLENGTH];
```

```
 91        int len;            // Length of input string.
 92        fin.getline( buf, BUFLENGTH, ',' );
 93        len = fin.gcount();
 94        fin >>storeCode >>quantity >>price;
 95        if (fin.fail()) return false;
 96        partName = new char[len];
 97        strcpy( partName, buf );
 98        return true;
 99    }
100
101    //-------------------------------------------------------------------------------
102    void
103    Part::print( ostream& fout ){
104        fout <<left  <<setw(25) <<setfill('.') <<partName;
105        fout         <<setw(3)  <<setfill(' ') <<storeCode;
106        fout <<right <<setw(5)                 <<quantity;
107        fout <<fixed <<setw(8)  <<setprecision(2) <<price;
108    }
```

Notes on `Part::read()`. This function attempts to read the input for a single part, 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.

- Line 92 reads characters from the input stream into the array named `buf`. The third parameter causes reading to stop at the first comma. Because `getline()` was used, that comma is removed from the stream 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 93 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.)

- Line 94 reads the three numbers that follow the name on the input line. Whitespace before each number is automatically skipped.

- Line 95 tests whether all inputs were properly found and stored. If not, no attempt is made to process the faulty line, and false is returned to the caller. The caller then handles the errors.

- Lines 96 and 97 happen only if all four inputs were read properly. They allocate storage, store the pointer in the `Part` instance, and copy the input into it.

- Allocation of space for the part name 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.

Notes on `Part::print()`. 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. Lines 103. . . 108 illustrate the use of the C++ formatting controls.

- 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 104–107 use `setw()` to control the width of the four columns. The field width must be supplied for every field.
- 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 104) 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 104 sets the fill character to `'.'`, so dot fill is used for the first column. Line 105 resets the filler to `' '`, so spaces are used to fill the other three columns.
- The price is a floating point number. Since the default format (%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.

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.