Applied C++
C++ Lessons to Accompany Applied C

Alice E. Fischer
University of New Haven

Table of Contents

Chapter 1: Issues and Overview

1.1 Why are we studying C++?

The introductory course at UNH is taught in plain C because C is a simpler language than C++ or Java, and the C compilers give simpler and clear error comments. Thus, C provides a more accessible platform for the beginner than the others. In addition, our programming courses serve Computer Engineers and Electrical Engineers as well as Computer Science and Cyber Systems students. C is a lower level language than the others and it provides a better platform for teaching about what computers are really like.

The UNH Data Structures is taught in C++ because C++ is a more modern language, and easier to use when modeling data structures. So students must make a transition from simple C to object-oriented programming in C++. The Intermediate C course seems to be the best time and place for that transition. Since we use a C-based textbook, these notes are provided as a C++ supplement.

1.1.1 Design Goals for C

C was designed to write Unix. C is sometimes called a "low level" language. It was created by Dennis Ritchie so that he and Kenneth Thompson could write a new operating system that they named Unix. The new language was designed to control the machine hardware (clock, registers, memory, devices) and implement input and output conversion. Thus, it was essential for C to be able to work efficiently and easily at a low level.

Ritchie and Thompson worked with small, slow machines, so they put great emphasis on creating an simple language that could be easily compiled into efficient object code. There is a direct and transparent relationship between C source code and the machine code produced by a C compiler.

Because C is a simple language, a C compiler can be much simpler than a compiler for C++ or Java. As a result, a good C compiler produces simple error comments tied to specific lines of code. Compilers for full-featured modern languages such as C++ and Java are the opposite: error comments can be hopelessly wordy and also vague. Sometimes, they do not correctly pinpoint the erroneous line.

Ritchie never imagined that his language would leave their lab and become a dominant force in the world and the ancestor of three powerful modern languages, C++, C#, and Java. Thus, he did not worry about readability, portability, and reusability. Because of that, readability is only achieved in C by using self-discipline and adhering to strict rules of style. However, because of the clean design, C became the most portable and reusable language of its time.

In 1978, Brian Kernighan and Dennis Ritchie published "The C Programming Language", which served as the only language specification for eleven years. During that time, C and Unix became popular and widespread, and different implementations had subtle and troublesome differences. The ANSI C standard (1989) addressed this by providing a clear definition of the syntax and the meaning of C. This standard was updated in 1999.

The result was a low-level language that provides unlimited opportunity for expressing algorithms and excellent support for modular program construction. However, it provides little or no support for expressing higher-level abstractions. We can write many different efficient programs for implementing a queue in C, but we cannot express the abstraction "queue" in a clear, simple, coherent manner.

1.1.2 C++ Extends C.

C++ is an extension and adaptation of C. The designer, Bjarne Stroustrup, originally implemented C++ as a set of macros that were translated by a preprocessor into ordinary C code. His intent was to retain efficiency and transparency and simultaneously improve the ability of the language to model abstractions. C++ improves C in many ways:

- Flexibility. C++ allows the programmer to define polymorphic types with more than one variation. A new class can be derived from an existing class. Data members no longer have unique types – they simultaneously have all the types above them in the polymorphic type hierarchy.

- A function no longer has a single unitary definition – it is a collection of one or more function methods, possibly in many classes, that operate on different combinations of parameter types.

- Operator definitions. A new method for an existing operator, such as + or *, can be defined to work on a programmer-defined type. This can greatly add to the readability of a program.

- Readability. Although `C` was retained as the basic vehicle for coding in `C++`, an application program, as a whole, may be much more readable in `C++` than in `C` because it is organized into meaningful, coherent modules, and both data members and function members are part of the same module

- Reusability. Code that is filled with details about a particular application is not very reusable. In `C`, the typedef and #define commands do provide a little bit of support for creating generic code that can be tailored to a particular situation as a last step before compilation. The `C` libraries even include two generic functions (`qsort()` and `bsearch()`) that can be used to process an array of any base type. However, `C++` provides much broader support for code reusability, in the form of template classes with type parameters. There is a substantial library of data-structure classes that are part of the language standard.

- Teamwork potential. `C++` supports highly modular design and implementation and reusable components. This is ideal for team projects. The most skilled members of the group can design the project and implement any non-routine portions. Lesser-skilled programmers can implement the routine modules using the expert's classes, classes from the standard template library, and proprietary class libraries. All these people can work simultaneously, guided by defined class interfaces, to produce a complete application.

1.2 Object Oriented Principles.

The term "object-oriented" has become popular, and "object-oriented" analysis, design, and implementation has been put forward as a solution to several problems that plague the software industry. OO analysis is a set of formal methods for analyzing and structuring an application from the application data's perspective, as opposed to the traditional functional or procedural point of view. The result of an OO analysis is an OO design. OO programs are built out of a collection of modules, often called *classes* that contain both function methods and data. Classes define data structures and the operations that can be performed on them. Access to all of the data and some of the method elements should only be through the defined methods of the class.

The way a language is used is more important in OO design than which language is used. `C++` was designed to support OO programming; it is a convenient and powerful vehicle for implementing an OO design. However, with somewhat more effort, that same OO design can usually be implemented in C. Similarly, a non-OO program can be written in `C++`.

Principles central to object-oriented programming are encapsulation, locality, coherent representation, and generic or polymorphic functions. The most fundamental OO design principles are:

- Encapsulation. A class should protect itself and take care of itself. The function members of a class should be the only functions to have full access to the class data. This is achieved by declaring member variables to be `private`. A member function can then freely use any data member, but outside functions cannot.

- Narrow Interface. Most methods are declared to be public, but some are not. Public visibility should only be used for methods that are part of the function's published interface. Methods intended only for internal use should be `private`.

- Initialization and Cleanup. One way a class takes care of itself is to define how class objects should be initialized. Initialization is done by *constructors*, which are like functions except that they have no return type. The name of the constructor is the same as the class name. A constructor is called automatically whenever a class object is declared or dynamically allocated. It uses its parameters to initialize the class's data members. A constructor might also dynamically allocate parts of the class object.

  Cleanup, in C++, is done by *destructors*. Each class has exactly one destructor that is called when the class object is explicitly freed or when it goes out of scope at the end of the code block that declares it.

The name of the destructor is a tilde (∼) followed by the class name. The job of a destructor is to free dynamically allocated parts of the class object.

- Coherent construction. Whenever possible, objects should be fully initialized when they are created. The data needed to initialize the object should be delivered as parameters to the class's constructor. The major exception to this guideline involves arrays of objects which cannot be fully initialized when the array is allocated.

1.2.1 Getting Started in C++

To write the first real C++ program, the programmer must understand I/O and classes. These topics are covered in Chapters 2 and 3. Chapter 3 gives an example program with no classes that a variety of input and output techniques. It defines an array of objects, opens and reads a file and prints the data.

At the end of Chapter 3 we introduce a program that will be carried through the following chapters, and developed more fully at each step. Chapter 3 gives a C version of this program, then an equivalent C++ version using one data class.

Chapter 4 introduces and implements some fundamental OO concepts and adds more functionality and a controller class to the program. Chapter 5 introduces dynamic allocation and an array that grows, as needed (the FlexArray class). Chapter 6 discusses STL strings and algorithms and uses the stl vector class, instead of a FlexArray. By mastering these C++ features, you should be well prepared to begin the Data Structures course.

Chapter 2: 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.

2.1 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 and for each member of a struct. 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 because you need to learn how to use it.

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.

2.2 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.

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.

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.h.
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.
```

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 `iostream` 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.

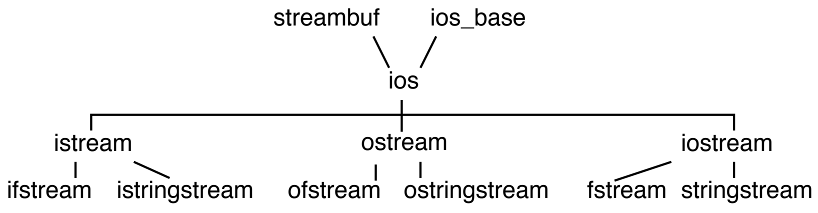


Figure 2.1: The stream type hierarchy.

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 3.2 shows how to test for hardware and format errors as well as eof.

2.3 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 2.5. 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.

2.4 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);`

2.5 End of File and Error Handling

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.

This topic gets very technical. The best approach is to skim the program fragments now and remember where to find them later when you have a problem and need help.

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.

```
51   Using get to read entire lines.
52   0 = 1:0:0:0: First line.
53   0 = 1:0:0:0: Second line.
54   0 = 1:0:0:0: Third line, with a newline on the end.
55   6 = 0:1:1:0: ----------------------------------
```

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

```
56   Using get to read entire lines.
57   0 = 1:0:0:0: First line.
58   0 = 1:0:0:0: Second line.
59   2 = 0:0:1:0: Third line, without a newline on the end.
60   ----------------------------------
```

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.

```
61   //===============================================================================
62   // The getline function removes the trailing \n from the stream and discards it.
63   // A. Fischer, April 2001                                        file: getline.cpp
64   //-------------------------------------------------------------------------------
65   #include "tools.hpp"
66
67   void use_getline( istream& instr )
68   {
```

```
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. }
}
```

2.5.1 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.

2.5.2 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 );
```

Chapter 3: Classes in C++

3.1 Classes are Structures with Brains

A class is like a C struct with added behaviors. Members of the class can be data, as in a C struct, or functions that operate on the class data members. Putting these functions inside the class declaration organizes the code into meaningful modules and simplifies a lot of the syntax for using the data members. This chapter gives a brief introduction to the most important aspects of classes in C++.

When we mean to build, we first survey the plot then draw the model.
. . . Shakespeare, King Henry IV.

3.1.1 Concepts and Terminology

Definitions and Conventions.

- The word *class* is used for the type declaration, very much like "struct" in C.
- An *object* is an *instance* of a class. We declare
- Class names normally start with an upper case letter. Object and function names start with lower case. Both use camelCase if the identifier is multiple words.

For example, suppose you have a class named *Student*. When a Student is created, it has all the parts defined by the class Student and we say it is an *instance* of Student. You might then have instances named Ann, Bob, Dylan, and Tyler. We can say that these people belong to the class Student, or have the type Student.

Common Uses for a Class. There are many kinds of classes. The most important are:

- A data class models a real-world object such as a ticket or a book or a person. We say it stores the *state* of that object.
- A container, or collection class, stores a set or series of data objects.
- A controller class is the heart of an application. It implements the logic of the business or the game that is being built. It often has a container object as a data member. One or more data objects are either in the container or in the controller class itself.

There is no rule about what should and should not be a data member of a class. Generally, the data members you define are those your code needs to store the data, do calculations, manage the files, and make reports.

Examples of data classes:

- TheaterTicket: Data members would be a the name of a show, a location, a date, a list price, a seat number, and a sold/available flag.
- Book: The title, author, publisher, ISBN number, and copyright date.
- Car: The owner, VIN, color, make, model, and year.

3.1.2 Function Members of a Class

The functions in a class define the way that class behaves. A class function can "see" and change the data members. Normally, functions are defined to do the routine things that every type needs: initialization and I/O. Class functions fall into these general categories:

- Constructors and destructors. A constructor initializes a newly-created class instance. A destructor frees dynamic memory that is part of the instance, if any exists.
- I/O functions. An I/O function is responsible for reading the data for one item from a file or formatting and printing the data from one item to a file or to the screen. Every class should provide a print function that outputs all of its data members. These are almost essential for debugging.
- Calculation functions. These do useful work. They are highly varied and depend on the application.
- Accessor functions. Accessors allow limited access by any outside class to the private members of current class.

Constructors A constructor has the same name as its class, and a class often has multiple constructors, each with a different set of parameters. Every time the class is instantiated, one of the constructors is called to initialize it.

Typically, a programmer provides a constructor with parameters that moves information from each parameter into the corresponding data member. It also initializes other data members so that the object is internally consistent and ready to use[1].

If you do not provide a constructor for the class, the compiler will provide a *null default constructor* for you. A default constructor requires no parameters. A null constructor does no initializations.

Destructors There is only one destructor in a class. It is run each time a class instance goes out of scope and "dies". This happens automatically whenever control leaves the block of code in which the object was instantiated. Many programs create and use a temporary object inside a block of code. At the end of the block, the object's destructor is called and the object is deallocated.

If part of an object was dynamically allocated, the programmer is responsible for managing the dynamic memory. The destructor of the class that includes the dynamic part should call *delete*, which calls the destructor, which causes the memory to be freed.

3.1.3 Encapsulation

One of the most important properties of a C++ class is that it can *encapsulate* data. Any class member can be defined to be either public or private. Public members are like global variables – any function anywhere can change them. Private members are visible only to functions in the same class[2].

When the data members are private, the class is able to control access to them. It also has the responsibility of providing public functions to carry out all the important actions that relate to the class: validation, calculation, printing, restricted access, and maybe even sorting. A class becomes the expert on its own members.

Delegation Delegate the activity to the expert.

Accessor functions. There are two kinds of accessors: getters and setters. A getter gives read-only access to otherwise private data, and is often given a name such as getCount() or getName(). A data class will often have one or more getters. However, it is a bad idea to provide a getter for every data member. Doing so is evidence of poor OO design.

Setters allow an outside function to modify the private parts of a class instance. This is rarely necessary and can usually be avoided by proper program design. There is no reason to provide a setter for every data member.

[1]More advanced programmers sometimes provide copy constructors and move constructors.
[2]Later, you will also learn about protected variables.

3.1.4 Instantiation

Instantiation is the act of creating an object, that is, an instance of a class. This can be done by two methods:

- Declaration creates *automatic* storage. When you declare a variable of a class type, space is allocated for it on the run-time stack. That space will remain until control leaves the block of code in which the object was declared. At block-end time, the stack frame is deallocated and the destructors will be run on all objects in the frame. No explicit memory management is needed, or permitted, on these automatic variables.

- Dynamic allocation. At any time during execution, a program can call *new* to instantiate a class. The result is a pointer to an object. The program is then responsible for explicitly freeing this object when it is no longer needed. For example, assume we have a class named Student. Then the following code allocates space to store the Student data, calls the 3-parameter Student constructor, and stores the resulting pointer in `newStu`: `newStu = new Student("Ann", "Smith", 00256987);`

  This dynamic object can be passed around from one scope to another and will persist until the program ends. It will probably be put into a container class. Suppose that later, when this data is no longer needed, it is attached to a pointer named `formerStu`. Then deallocation would look like this: `delete formerStu;`

3.2 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.

```
1   //  main.cpp
2   //  Store
3   //  Demo program for basic input and output. ---------------------------------
4   //
5   //  Created by Alice Fischer on 7/17/15.
6   //  Michael and Alice Fischer, August 11, 2014
7
8   #define N 1000          // Maximum number of parts in the inventory
9   #include "Part.hpp"
10
11  long readParts( istream& fin, Part* data );
12  void printAll( ostream& fout, Part* inventory, long n );
13
14  //---------------------------------------------------------------------------
15  int main( void )
16  {
17      Part inventory[N];
18      ifstream instr( "parts.in" );
19      if ( !instr )  fatal( "Cannot open parts file %s", "parts.in" );
20
21      cerr << "\nReading inventory from parts input file.\n";
22      const long n = readParts( instr, inventory );
23      instr.close();
24
25      cerr <<n <<" parts read successfully.\n\n";
26      printAll( cout, inventory, n );
27      bye();
28      return 0;
29  }
```

```
30
31  //---------------------------------------------------------------------------
32  long readParts( istream& fin, Part* data ) {
33      Part* p = data;     // Cursor to traverse data array.
34      Part* pend = p+N;   // Off-board pointer to end of array.
35
36      while (p<pend) {
37          fin >> ws;
38          bool gotData = p->read( fin );
39          if (gotData) {              // First handle the normal case
40              ++p;                    // Position cursor for next input.
41              if (fin.eof()) break;   // no newline at EOF
42          }
43          // No good data on current line
44          // EOF with gcount()==0 means normal EOF with newline
45          else if (fin.eof() && fin.gcount()==0) break;
46          else {                      // Partial line or data conversion error
47              cerr <<"Error reading line " <<(p-data+1) <<"\n";
48              fin.clear();
49              fin >>flush; // Skip rest of defective line.
50          }
51      }
52      return p - data;  // Number of data lines read correctly and stored.
53  }
54
55  //---------------------------------------------------------------------------
56  void
57  printAll( ostream& fout, Part* inventory, long n ){
58      Part* p = inventory;
59      Part* pend = inventory+n;
60
61      for ( ; p<pend; ++p) {
62          p->print( fout );
63          fout<<endl;
64      }
65  }
```

Inventory: The main program.

- Line 8: The constant `N` is defined here because it is required by `readParts` and by the main program.
- Line 18 declares and opens the input stream; line 23 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.
- A long integer variable is declared in the middle of the code on line 22. This is legal in C++. It is appropriate if the variable is used only in the immediate neighborhood of the declaration.
- The call on `bye()` on line 27 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 (lines 38 and 62). The main program contains the functions that read and print many parts.

Reading the file into the inventory array. (Lines 31 . . . 53)

- 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 scanning pointer is set to the beginning, and will walk down the array as the input loop progresses, pointing at each array slot sequentially. The end pointer points to the slot past the last *allocated* array slot. When processing is finished, 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 40, but only if the program received good data.
- Line 52 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 36, `while (p<pend)`) or until the end of the input file is reached. (The test is on line 41).
- 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 37 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 input extraction 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 38 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 39 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 45 is the normal eof test that ends processing for files that end properly with a newline.
- Lines 48. . . 49 handle errors by clearing the stream's status flags and removing the rest of the defective input line from the stream. flush is defined in the tools library.
- DO NOT put your end-of-file test inside the `while` test on line 36.

Printing the inventory array. (Lines 55. . . 65)

- Lines 55 and 56 set pointers to the beginning and end of the DATA in 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 (left) and the screen output (right).

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.

```
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
```

```
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 class declaration: `part.hpp`**.**

```
66  //----------------------------------------------------------------------------
67  //  Header file for hardware store parts.                             Part.hpp
68  //  Created by Alice Fischer on July 17, 2015, from Mon Dec 29 2003.
69  //
70  #include "tools.hpp"
71  #define BUFLENGTH 100   // Maximum length of the name of a part
72
73  //----------------------------------------------------------------------------
74  class Part {
75  private:
76      string partName;
77      int storeCode, quantity;
78      float price;
79  public:
80      bool read( istream& fin );
81      void print( ostream& fout );
82  };
```

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

```
83  //------------------------------------------------------------------------------
84  //  Implementation file for hardware store parts.                       Part.cpp
85  //  Created by Alice Fischer on July 17, 2015, from Mon Dec 29 2003.
86  //
87  #include "Part.hpp"
88  //------------------------------------------------------------------------------
89  // Preconditions: fin is open for input; no initial whitespace; eof flag is off;
90  bool
91  Part::read( istream& fin ) {
92      char buf[BUFLENGTH];
93      fin.getline( buf, BUFLENGTH, ’,’ );
94      fin >>storeCode >>quantity >>price;
95      if (fin.fail()) return false;
96      partName = string(buf);
97      return true;
98  }
99
100 //------------------------------------------------------------------------------
101 void
102 Part::print( ostream& fout ){
103     fout <<left  <<setw(25) <<setfill(’.’) <<partName;
104     fout         <<setw(3)  <<setfill(’ ’) <<storeCode;
105     fout <<right <<setw(5)                 <<quantity;
106     fout <<fixed <<setw(8)  <<setprecision(2) <<price;
107 }
```

Notes on the Part module.

- Line 70: 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;`.
- Line 74: In C++, you can use `struct` or `class` 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.
- Lines 75. . . 78: The data members of a class are the parts that you would declare for a C `struct`. They are virtually always made private.
- Lines 79. . . 81: 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.

- Lines 80. . . 81: Note the ampersands in these 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.
- Line 82: A class declaration ends in a close-brace that matches the one on the first line of the class. The semicolon after the brace is necessary. Omitting it produces strange and confusing compiler error comments.

Notes on `Part::read()`. This function is called from the readParts function() in the main module. It attempts to read the input for a single part and checks for errors. If an error is discovered while reading a data set, control and an error code return to main, where the entire data set is skipped and the input stream is flushed up to the next newline character. The faulty input is reported.

- Line 93 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 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.
- Line 96 allocates space for the new part name and copies it into the Part array.

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, using 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. . . 106 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 103–106 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 103) 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 103 sets the fill character to `'.'`, so dot fill is used for the first column. Line 104 resets the filler to `' '`, so spaces are used to fill the other three columns.
- Line 106: 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. The output without a final newline is the same as was shown, above, for a file lacking a final newline as 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 were covered in the the prior chapter.

3.3 Example Programs

In this section are two versions of the same program. The first is written in C, the second in C++, with one class. The next chapters present three more versions in C++: one with two classes and more functionality, one that uses an STL vector, and one that uses dynamic allocation. Together, these four C++ versions introduce most of the fundamentals of C++ syntax and some basic OO design principles. Notes are given after every part of the code to tell you what to look for.

3.3.1 Buying Tickets: the C version

```
//
//  main.c
//  Tickets
//
//  Created by Alice Fischer on 7/13/15.
//  Copyright (c) 2015 Michael and Alice Fischer. All rights reserved.
//

#include "tools.h"
#define BASE_PRICE 2.25

typedef struct {
    int quantity;
    int type;
    float price;
} Ticket;

//-----------------------------------------------------------------------------
const cstring ageLabels[] = { "Adult  ","Child  ","Senior " };
const float adultPrice  = BASE_PRICE;
const float childPrice  = (50 * BASE_PRICE) / 100 ;
const float seniorPrice = (80 * BASE_PRICE) / 100 ;

void getTicketOrder( char* ageCode, int* quantity );
Ticket makeTicket( int quant, int typ, float cost );
void printTicket( Ticket t );

```

```c
//-------------------------------------------------------------------------
int main( void ) {

    char ageCode;
    int quantity;
    Ticket tik;

    puts( "\nBus Ticket Vending Machine\n" );
    getTicketOrder( &ageCode, &quantity );
    tik = makeTicket( quantity, ageCode, adultPrice );
    printTicket( tik );
    puts( "\nPlease swipe your credit card, then take your tickets.\n");
    return 0;
}

//-------------------------------------------------------------------------
void getTicketOrder( char* ageCode, int* quantity )
{
    int quant;      // Number of passengers, local.
    char choice;    // The age code: 'a', 'c', or 's', local.
    char temp;      // to read in the age code. local.
    char* found;    // Status return from validation of menu entry.

    printf( "    a: Adult %.2f\n    c: Child under 12  %.2f\n    s: Senior Citizen %.2f\n", adultPrice, chi

    // Dispense bus tickets, adult, child, or senior.
    for(;;){
        puts( "Please select a, c, or s: " );
        scanf( "%c", &temp );
        choice = tolower( temp );
        found = strchr ("acs", choice);
        if ( found != NULL ) break;
        cleanline( stdin );   // Discard chars to end of line.
        printf("Age selection must be a, c, or s; you entered: %c\n", choice);
    }
    for(;;){
        printf( "Please enter the quantity: " );
        scanf( "%i", &quant );
        if (quant>0 && quant<10) break;
        cleanline( stdin );    // Discard chars to end of line.
        printf( "Quantity must be between 1 and 10 " );
    }
    *ageCode = choice;
    *quantity = quant;
}

//-------------------------------------------------------------------------
Ticket makeTicket( int quant, int typ, float cost ){
    Ticket t;
    t.quantity = quant;
    t.type = typ;
    t.price = cost * quant;
    return t;
}

//-------------------------------------------------------------------------
void printTicket( Ticket t ){
    printf( "You bought %i %s tickets for $%.2f\n",
        t.quantity, ageLabels[t.ageCode], t.price);
}
```

This program is written using several C techniques covered in recent chapters. Notable elements include:

- A global constant, #defined at the top because it might need to be changed frequently.
- A typedef for a struct type.
- Several global const variables. Global variables are a bad idea. Global constants are OK.
- MakeTicket returns a struct, and printTicket has a struct parameter.
- Main is brief. A main program should always be brief. This one prints a heading and does its job by calling other functions.
- The getTicketOrder function reads and validates two data values. One of them is a menu choice of type char.
- tolower() or toupper() should always be used on an input code before validation or use in a switch.
- Validation is done by using strchr() to search a string of valid input codes for the code the user typed. A non-null search result indicates a valid input.
- The getTicketOrder function uses pointer parameters to return the two data values it read. The last lines before returning from the function store the inputs in the caller's variables.
- The cleanline() function from the tools library is used after an error to clear out possible junk on an input line. This leaves the stream in a predictable state, and ready for a new input.
- The makeTicket() function allocates space for a ticket, initializes that space, and returns it.
- printTicket() formats the information in the struct for printing.

3.3.2 Buying Tickets: the first C++ version

This version introduces the first C++ class and illustrates many of the OO elements that were defined above. In a C program, main() does its work by calling a series of functions. In a C++ program, main() instantiates the application's major class, then calls its functions.

main()

```
//-----------------------------------------------------------------------------
//  main.cpp
//  Tickets
//
//  Created by Alice Fischer on 7/13/15.
//  Copyright (c) 2015 Alice Fischer. All rights reserved.
//

#include "tools.hpp"
#include "ticket.hpp"
//-----------------------------------------------------------------------------

int main( void ) {
    char ageCode;
    int quantity;

    cout <<"\nBus Ticket Vending Machine";

    Ticket::getOrder( ageCode, quantity );
    Ticket tik( ageCode, quantity );

    cout << endl;
    tik.print( cout );
    cout <<"\nPlease swipe your credit card, then take your ticket.";
    return 0;
}
```

Ticket.hpp. A C++ header file defines the data members and prototypes of the class. Sometimes it contains a related type declaration. The purpose of the Ticket class is to gather together all of the information about tickets: the types of tickets, prices, valid ticket requests, and how to format a ticket for printing. Having all the related data and functions in one place makes a program easier to write and easier to maintain. It also simplifies function calls in several ways.

```
27  //-----------------------------------------------------------------------------
28  //  ticket.hpp
29  //  Tickets, version with one class.
30  //
31  //  Created by Alice Fischer on 7/13/15.
32  //  Copyright (c) 2015 Alice Fischer. All rights reserved.
33  //
34
35  #ifndef TICKET
36  #define TICKET
37
38  #include "tools.hpp"
39  //-----------------------------------------------------------------------------
40
41  class Ticket {
42    private:
43      char ageCode;           // a=adult, c=child, s=senior
44      string ageLabel;        // The word "adult" or "child" or "senior"
45      int quantity;           // Number of passengers on this ticket.
46      float ticketPrice;      // Price of this ticket.
47
48    public:
49      Ticket( char age, int quantity  );
50      ~Ticket(){};
51
52      static void getOrder( char& ageCode, int& quant );
53
54      void print( ostream& out );
55  };
56
57  #endif // ifndef Ticket
```

Include guards. Lines 35, 36, and 57 are the *include guards*. In a multi-module application, these are important to ensure that no header file gets included twice in the same compile module. The symbol, `TICKET` used in the include guards is normally written in all-upper-case, but can otherwise be the same as the class name. Some programmers use longer multi-part names, with underscores. The important thing is that the symbol is unique within the application.

Data members. In this Ticket class, the data members correspond to the members of the struct in the C version. As always, all of the data members are private. Note that we are using the C++ `string` type instead of the simple C `char*`. Strings are more complex but have important advantages that will be explained in Chapter 5.

The functions are also parallel, but note that the word "Ticket" no longer needs to be part of the name of each function because everything here relates to tickets and all names are qualified by being inside the Ticket class.

Static functions. Line 52 is a static class function that prompts the user to input the data for a new ticket. It is static so that it can be called before a Ticket object is created. To call a static function, use the class name and `::`, as shown on line 19.

Non-static functions. Line 54 corresponds to the printTicket() function in C, but this one is written so that the output could be sent to any output stream, not only to the standard output stream. To call an ordinary (non-static) function, use the name of a class instance, as shown on line 23.

Constructor and destructor. Line 50 is the destructor. It has empty braces because there is no dynamic memory to free in this class. Line 49 is the prototype of the class constructor. The constructor corresponds to the init() function in C, but there are differences: constructors never return anything, and they use memory that was allocated prior to calling the constructor. The job of a constructor is only to initialize that memory.

C init() function	C++ constructor
Allocates an object in stack memory	Does not allocate stack memory
Returns the object	Does not return anything
Has no parameters	Often has parameters
Initializes the fields of the object	Initializes the fields of the object using the parameters but it does not allocate stack memory and it does not return anything.

Ticket.cpp

```
 58  //---------------------------------------------------------------------------
 59  //  ticket.cpp
 60  //  Tickets, version with one class.
 61  //
 62  //  Created by Alice Fischer on 7/13/15.
 63  //  Copyright (c) 2015 Alice Fischer. All rights reserved.
 64
 65  #include "ticket.hpp"
 66  //---------------------------------------------------------------------------
 67  // This machine sells tickets at a fixed base price.  Tickets for children
 68  //     are half price, and seniors are 80%.
 69  #define BASE_PRICE 2.25
 70  const float adultPrice  = BASE_PRICE;
 71  const float childPrice  = round( 50 * BASE_PRICE) / 100 ;
 72  const float seniorPrice = round( 80 * BASE_PRICE) / 100 ;
 73  const string codes = "acs";  // Menu choice character codes
 74
 75  //---------------------------------------------------------------------------
 76  // Postcondition: the output parameters are valid.
 77  void
 78  Ticket:: getOrder( char& ageCode, int& quant ) {
 79      char temp;      // to read in the age code. local.
 80      size_t found;
 81      cout <<fixed  <<setprecision(2)
 82          <<"\n    a: Adult:           $" << setw(5) << adultPrice
 83          <<"\n    c: Child under 12: $" << setw(5) << childPrice
 84          <<"\n    s: Senior Citizen: $" << setw(5) << seniorPrice <<"\n";
 85
 86      // Take an order for bus tickets, adult, child, or senior.
 87      for(;;){
 88          cout << "Please select a, c, or s: " ;
 89          cin >> temp;
 90          ageCode = tolower( temp );
 91          found = codes.find_first_of(ageCode);
 92          if ( found != string::npos ) break;
 93          cleanline(cin);   // Discard chars to end of line.
 94          cout <<"Age selection must be a, c, or s; you entered: " <<ageCode <<endl;
 95      }
 96      for(;;){
 97          cout << "Please enter the quantity: ";
 98          cin >> quant;
 99          if (quant>0 && quant<10) break;
100          cleanline(cin);   // Discard chars to end of line.
101          cout <<"Quantity must be between 1 and 10 \n";
102      }
103  }
104
```

```
105  //------------------------------------------------------
106  void Ticket::print( ostream& out ){
107      out << "You bought " <<quantity <<" " << ageLabel
108          <<" seat(s) for $ " << setw(6) <<right << ticketPrice;
109  }
110
111  //-----------------------------------------------------------------------
112  // Precondition: getOrder() has been called and the age code has been validated.
113  //
114  Ticket::Ticket( char age, int quantity ){
115      float singlePrice = 0;
116
117      ageCode = age;
118      switch (ageCode) {
119          case 'a':
120              singlePrice = adultPrice;
121              ageLabel = "Adult ";
122              break;
123          case 'c':
124              singlePrice = childPrice;
125              ageLabel = "Child ";
126              break;
127          case 's':
128              singlePrice = seniorPrice;
129              ageLabel = "Senior";
130              break;
131          default:  fatal("ageCode precondition has been violated.");
132      }
133      this->quantity = quantity;
134      ticketPrice = singlePrice * quantity;
135  }
```

This file contains implementations of all the class functions that are longer than one line. You can see that most of the code from C is here somewhere. However, there are changes.

The include command (line 65).

- Every .cpp file must include its own header file.
- No other header files should be included by the .cpp; all other necessary `#includes` should be in the .hpp.

The constants (lines 69–73).

- Constants are data objects. Data objects are never defined in header files. They must be in the .cpp file because header files are typically included more than once in an application. For example, ticket.hpp is included by *both* ticket.cpp and main.cpp. Its purpose is to communicate to main all the type information that is needed to compile calls in main() on functions in the Ticket class.
- If a data object is included more than once in an application, the linker will fail when it tries to combine the modules of the application. Include guards *do not* protect against this. They only protect against including the same info in a *single* compile module.
- `#define` is used for the first constant, which is simply a number.
- The next three are const variables, defined by constant expressions, that is, expressions that can be evaluated by the compiler at compile time.
- The last constant, `codes` is a string consisting of all the legal menu choices. It will be used for validating the user's menu choice.
- The C++ `string` class easier to use than a C char* because the user does not need to worry about storage management for the data. The `string` class provides an extensive library of useful functions and takes care of all allocation and deallocation issues. Thus, it is convenient and safe, but somewhat more costly in both time and space than a `char*`.

- The call on `round()` (lines 71 and 72) rounds a number to the closest penny. This is needed because half of the base price, and 80% of the base price might have fractional pennies.

The constructor (lines 112-135).

- The constructor takes two parameters and uses them to initialize four data members. Two of the data members are initialized in the switch statement, based on the first parameter.
- The constructor has a local variable. This is a local variable (instead of a class member) because its use is strictly local. No other function needs the information stored in it. Most class functions have their own local variables.
- Line 133 initializes a data member that has the same name as a parameter. The keyword `this->` refers to the object that is being created, and the keyword is required to break the ambiguity.
- The other parameter has a name that is different from the data member, so no `this->` is required.
- It is common to use the same name, even though you need to use `this->`. Some experts consider it less confusing if the names are the same.

The input and output functions (lines 75–110).

- All the input (in `getOrder()`) and output (in `print()`) is done using the C++ streams and syntax. Compare this to the C version. Each one has its advantages and disadvantages.
- Lines 81–84 in `getOrder` do a nice job of formatting the menu. Note the use of the stream manipulator `setw()`. Some manipulators (right) remain set until changed. Others, such as setw(), must be written for each field.
- To make it easy to validate, the input is converted to lower case when it is read (lines 89 and 90).
- Line 91 validates the menu choice. It uses the function `find_first_of()` from the C++ stl string class.
- The call on `find_first_of()` searches for the `ageCode` character in the string `"acs"` (initialized on line 73 to a list of all legal menu choices). The result (tested on line 92) is the subscript of the character in the string if the input is found, or `string::npos` if the input is not found.
- An invalid menu choice could be a simple typo, or it could happen from a kitten walking on the keyboard or the user going to sleep on it. There is no guessing. After a failed input, it is a very good idea to be sure that there is nothing left on the input line, so that the user starts afresh.
- Validating the numeric input (line 99) is easier than validating a character. Otherwise, the second input loop is structured like the first.
- The `print()` function (lines 106–109) does a nice job of formatting the output. Note the use of the stream manipulators `right` and `setw()`.

Chapter 4: Arrays in C++

4.1 Concepts and Syntax

C++ arrays.

- In C++, an *array* is declared and used very much the same way as a C array.
- A C++ array can be defined that stores any type of object: a primitive type such as `double`, a pointer type, or a class type such as `Ticket`.
- You can have an array of objects or an array of pointers to objects. (Unlike Java). Dyamically allocated objects (created with new) would be stored in an array of pointers.
- An array declaration looks very much the same as it does in C. Here are some examples:
  ```
  float money[ 10 ];
  Ticket purchase[ 10 ];
  Ticket* purchase[ MAX ]; // For dynamically allocated Tickets.
  ```
- Arrays can be initialized in the declaration, as they are in C.
  ```
  float money[ 10 ] = {};  // Initialize all 10 slots to 0.0
  Ticket purchase[ 10 ];   // Initialize all slots to a default ticket.
  ```
- To create the array of default objects, the class must supply a default constructor, that is, a constructor with no parameters.

4.1.1 Example: Two Classes and an Array of Objects

In this section, we give the second version of the C++ Ticket program. It defines two classes and uses an array of objects. The main program is given first, followed by two modules, Vendor and Tickets. Tickets is a data file (model file) as it was in version 1 of this program. Vendor is a controller class: its primary function is called from main, and it controls the path of execution until the program is finished. Program notes are given after each file.

Compare this version to TicketsV1 in the prior chapter. You will see that most of the code is the same, but it has been rearranged according to the most basic OO design principle:

> Expert. A class should be the expert on its own members. It should manage them, protect them, and take care of emergencies involving them.

Tickets-V2: The main module.
This is a typical main module 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 preprocessor commands.

- Line 9: The tools header is included because main() uses cout and `<<`.
- Line 10: The vendor header is included because main() instantiates the Vendor class and calls two of its functions.
- Line11: The constant `BASE_PRICE` is defined here because it is required by both the Vendor class and the Ticket class.

```
 1  //----------------------------------------------------------------------------
 2  //  main.cpp
 3  //  Tickets
 4  //
 5  //  Created by Alice Fischer on 7/13/15.
 6  //  Copyright (c) 2015 Michael and Alice Fischer. All rights reserved.
 7  //
 8
 9  #include "tools.hpp"
10  #include "vendor.hpp"
11  #define BASE_PRICE 2.25     // The current price of an adult ticket.
12  //----------------------------------------------------------------------------
13
14  int main( void ) {
15      cout <<"\nBus Ticket Vending Machine\n\n";
16
17      Vendor v( BASE_PRICE );
18      v.vend();
19      cout << endl;
20      v.print( cout );
21      cout <<"\nPlease swipe your credit card, then take your ticket.\n";
22      return 0;
23  }
```

The main function.

- Line 15 prints a title so that the user will know the program is in operation.
- Line 17 instantiates the controller class and line 18 calls its primary function.
- Lines 19–20. Control returns to main() when the vend() function exits. At that point, main() prints the results of the vending operation, followed by a closing message.

Vendor: the Controller Class Some of the code that was in the Ticket class in V1 is now in this class. The decision of where to put a function or constant is based on what class is the *expert* on that issue. (This will be explained further in the discussion below.) Now that we have a controller class, it becomes the expert on vending tickets, and all the code related to ordering tickets is somewhere in the Vendor class.

- We see the usual include guards and #include tools.hpp. The ticket header is included because this class creates Tickets and calls their functions.
- The constant `MAXTIK` is defined here because this class is the *expert* on how many tickets can be handled. (`MAXTIK` is used to define the length of the Ticket array.)

The Vendor class declaration: `Vendor.hpp`.

- There are eleven data members, all private, in three categories:
  - Constants (lines 44–48).
  - Information about the set of tickets (lines 56–58).
  - Internal variables used to communicate between functions of this class (lines 51–53).
- The constant data member (line 44). Some constants are known at compile time and can be initialized in the class declaration. However, some constants rely on information that is input at run time. C++ rules say that such constants must be initialized by *ctors*, written after the parameters and before the opening brace of the code. The list of ctors starts with a colon ":", followed by the name of the constant class member. The initial value for that member is written in parentheses. If there is more than one ctor, the colon is not repeated and the ctors are separated by commas.
- Line 56 declares an array of Ticket objects. To do so, the Ticket class must supply a default constructor. A default constructor that actually does nothing is defined on line 170.

- There are four public function members that, together, form the class interface:
  - This constructor (line 65) must initialize `adultPrice`, which is a constant class member. The ctor on line 65 does the job using the parameter to the Vendor constructor.
  - The destructor (line 66) has empty braces because this class does not have dynamic allocation.
  - `vend()` (line 67) is the controller's primary function. It is called from main and carries out the work of the application.
  - A print function (line 68). Every class should have a print function – it is really hard to debug without it.
- There are also two private function members that are called from other class functions but not from outside the class: `instructions` and `getOrder()`.

```
24  //---------------------------------------------------------------------------
25  //  vendor.hpp
26  //  TicketsV2: version with two classes.
27  //
28  //  Created by Alice Fischer on 7/13/15.
29  //  Copyright (c) 2015 Michael and Alice Fischer. All rights reserved.
30  //
31  #ifndef VENDOR
32  #define VENDOR
33
34  #include "tools.hpp"
35  #include "ticket.hpp"
36  #define MAXTIK 10
37
38  //---------------------------------------------------------------------------
39  class Vendor {
40    private:
41      //------------------------------------------------------------------------
42      // This machine sells tickets at a fixed base price.  Tickets for children
43      //     are half price, and seniors are 80%.
44      const float adultPrice;
45      const float childPrice = round( 50 * adultPrice) / 100 ;
46      const float seniorPrice = round( 80 * adultPrice) / 100 ;
47      const string codes = "acs";  // Menu choice character codes
48      const string ages[3] = {"adult","child","senior"};
49
50      // These members are used to communicate between class functions.
51      char ageCode;
52      int quantity;
53      size_t found;
54
55      // These members store information about the customer's ticket order.
56      Ticket purchase[MAXTIK];
57      int nTiks = 0;      // Number of tickets currently in the collection.
58      float totalPrice = 0;
59
60      // These functions are used only by other class functions.
61      void getOrder();
62      void instructions( ostream& out );
63
64    public:
65      Vendor( float price ) : adultPrice( price ){}
66      ~Vendor(){}
67      void vend();        // dispense a set of tickets.
68      void print( ostream& out );
69  };
70  #endif // ifndef VENDOR
```

Notes on Vendor.cpp

- `instructions()` is a brief function used to remove a layer of detail from the `vend()` function. It is private because there is no need for it to be visible outside this class.
- The `getOrder()` function is exactly parallel to V1's `getOrder()`, but now it is in the Vendor class instead of in the Ticket class.
- The `print()` function is entirely new. It has a loop to print multiple tickets and delegates the task of printing each ticket to Ticket::print().
- The `vend()` function gathers together parts of the application that had been scattered in V1. It contains code (line 142) that was in main() in V1, and some lines (136–141) that were in the Ticket constructor. New code is also introduced (lines 134 and 143–147) to handle multiple tickets and the total price for multiple tickets.

Vendor.cpp

```
 71  //----------------------------------------------------------------------------
 72  //  vendor.cpp
 73  //  TicketsV2:  version with two classes.
 74  //
 75  //  Created by Alice Fischer on 7/13/15.
 76  //  Copyright (c) 2015 Michael and Alice Fischer. All rights reserved.
 77  //
 78
 79  #include "vendor.hpp"
 80  //----------------------------------------------------------------------------
 81  // Postcondition: the age code and quantity are valid.
 82  void
 83  Vendor::getOrder() {
 84      char temp;      // to read in the age code. local.
 85
 86      cout <<fixed  <<setprecision(2)
 87          <<"\n    a: Adult:          $" << setw(5) << adultPrice
 88          <<"\n    c: Child under 12: $" << setw(5) << childPrice
 89          <<"\n    s: Senior Citizen: $" << setw(5) << seniorPrice <<"\n";
 90
 91      // Take an order for bus tickets: adult, child, or senior.
 92      for(;;){
 93          cout << "Please select a, c, or s: " ;
 94          cin >> temp;
 95          ageCode = tolower( temp );
 96          found = codes.find_first_of(ageCode);
 97          if ( found != string::npos ) break;
 98          cleanline(cin);   // Discard chars to end of line.
 99          cout <<"Age selection must be a, c, or s; you entered: " <<ageCode <<endl;
100      }
101      for(;;){
102          cout << "Please enter the quantity: ";
103          cin >> quantity;
104          if (quantity>0 && quantity<10) break;
105          cleanline(cin);   // Discard chars to end of line.
106          cout <<"Quantity must be between 1 and 10 \n";
107      }
108  }
109
110  //----------------------------------------------------------------------------
111  void Vendor :: instructions( ostream& out ) {
112      out << "Please enter your purchases one at a time.  When you are finished "
113          << "the total price will be charged to your charge card.\n";
114  }
115
```

```
//----------------------------------------------------------------------------
void
Vendor :: print( ostream& out ){
    out <<fixed <<setprecision(2) <<"You bought the following tickets: \n";
    for ( int k=0; k<nTiks; ++k )  {
        purchase[k].print( cout );
        cout << endl;
    }
    out << "Your credit card will be charged $" <<setw(6) << totalPrice <<endl;
}

//----------------------------------------------------------------------------
void
Vendor :: vend(){
    float price;
    char answer = 'y';

    instructions( cout );
    do {
        getOrder();
        switch (ageCode) {
            case 'a': price = adultPrice; break;
            case 'c': price = childPrice; break;
            case 's': price = seniorPrice; break;
            default : price = 0;
        }
        Ticket tik( ages[found], quantity, price );
        totalPrice += tik.getPrice();
        purchase[nTiks++] = tik;   // Shallow copy, ok because no dynamic parts.
        cout << "Do you want more tickets (y, n)?  ";
        cin >> answer;
    } while (tolower(answer) == 'y' && nTiks < 10);
    cout << "Purchase has been concluded.\n";
}

```

The header for the Ticket class: `ticket.hpp`**.**

```
//----------------------------------------------------------------------------
//  ticket.hpp
//  Tickets, version with one class.
//
//  Created by Alice Fischer on 7/13/15.
//  Copyright (c) 2015 Michael and Alice Fischer. All rights reserved.

#ifndef TICKET
#define TICKET

#include "tools.hpp"
//----------------------------------------------------------------------------
class Ticket {
  private:
    string ageLabel;        // The word "adult" or "child" or "senior"
    int quantity;           // Number of passengers on this ticket.
    float ticketPrice;      // Price of this ticket.

  public:
    Ticket( string age, int quantity, float price );
    Ticket (){};
    ~Ticket(){};

    float getPrice() const { return ticketPrice; }
    void print ( ostream& out ) const;
};
#endif // ifndef Ticket
```

The class declaration for Tickets in V2 is similar to the V1 declaration with small changes:

- One data member is gone! `ageCode` is no longer needed because Vendor (not Ticket) is now the expert on age codes.
- Line 170: The constructor has an additional parameter, the price because Vendor is the expert on prices.
- Line 171: a default constructor is added, needed to create an array of Tickets.
- Line 175: a `getPrice()` function has been added to allow the Vendor class to calculate the total price of a set of tickets.

The implementation file, `ticket.cpp`.

- The constants in V1's ticket.cpp became data members in Vendor (lines 44–47) because they concern the price of all tickets, not the price of a single ticket. Similarly, the `getOrder()` function was moved to Vendor.
- Ticket.cpp is now very short: it contains only the ticket constructor and the print function.
- The constructor is shorter and simpler because the correct price is supplied as a parameter.

```
//-----------------------------------------------------------------------------
//  ticket.cpp
//  Tickets, version with one class.
//
//  Created by Alice Fischer on 7/13/15.
//  Copyright (c) 2015 Michael and Alice Fischer. All rights reserved.

#include "ticket.hpp"

//-----------------------------------------------------------------------------
// Precondition: getOrder() has been called.
//
Ticket::Ticket( string age, int quantity, float price  ){
    this->quantity = quantity;
    ageLabel = age;
    ticketPrice = price * quantity;
}

//-------------------------------------------------------------
void Ticket::print( ostream& out ) const {
    out << "You bought " <<quantity <<" " << ageLabel
        <<" seat(s) for  $" << setw(6) <<right << ticketPrice;
}
```

Chapter 5: Dynamic Allocation in C++

5.1 The Stack: Automatic Storage

At run time, in a C or C++ environment, there is a data structure called the runtime stack that contains storage for parameters and variables. The oldest variables are from `main()`, and they are at the bottom of the stack. Then `main()` calls functions that call other functions.

Each time a function is called, an area of storage called a *stack frame* is created (allocated and initialized) at the top of the stack. When the function returns, that same area is deallocated. It is called `automatic` storage because it automatically comes and goes when needed. A program does not need to explicitly request this storage, and it does not need to explicitly free it.

A stack frame contains the functions arguments, its local variables, and three management pointers that are used by the runtime system to make everything work. Frames go on and off the stack in LIFO order (Last In, First Out). Thus, when you return from a function, control is back in the environment from which the function was called.

Stack storage is the most efficient kind of storage in both time and space. It should be used wherever possible. However, it has limitations: the size of an array or object on the stack cannot change, it is fixed at compile time. This creates a serious problem: it is very common to be unable to predict how much storage will be needed because, often, a programmer cannot predict how much data there will be.

The most common and vexing example of this problem is string input. If you ask a user to input a name (or anything else alphabetic), he might enter a name longer than the array you allocated to store it. If this is done blindly, the program will begin *walking on storage*, that is, overwriting the values in the adjacent memory locations. This event is also called *buffer overflow.* For this reason, every time a string is read into an array, the input command must limit the input to the size of the array. Buffer overflow is a problem that can be avoided. However, many many programmers never learn to handle input correctly and many many programs are buggy for this reason.

Some of the most famous computer viruses did and still do exploit buffer overflow in commercial code. Educating programmers to write safer code works, but far too many students do not get that education. A newer approach to safe coding is to provide system data structures and code libraries that will do safe reads. C++ provides the type `string` to solve this problem. A string in C is a pointer to a null-terminated array of characters. A string in C++ is much more. It is a class that has:

- A dynamically allocated array of char.
- An int, the current length of that allocation.
- A second int, the number of chars of data that are currently stored in the array.
- Functions to manage the storage, including making the allocation larger, as needed.
- Input, output, and assignment functions to use strings safely.

In the Hardware Store example from Chapter 3, we simplified the code by using the C++ `string` type to allow the part name to be any length needed. In this chapter, we convert the Vendor class to using dynamic storage so that it is unnecessary to limit the number of ticket orders a customer can make. To do this we introduce a third class to the example: a *container* class called FlexTiks. This class provides a growing array of Ticket objects, and demonstrates how growing arrays work.

The next chapter converts the Vendor program to use type `vector`, which (like `string`) is part of the C++ STL library. It provides a growing array in much the same way as FlexTiks but provides more functionality and is more general.

5.2 The Heap: Dynamic Storage

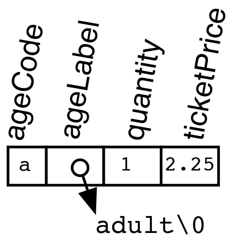
5.2.1 C++ syntax: new and delete

- `new` is the C++ analog of `malloc`.
- There are no C++ analogs of `calloc` and `realloc`.
- `delete` is the C++ analog of `free` for non-arrays.
- `delete[]` is used to delete arrays and all the objects in them.
- Everything you create with `new` in a class constructor or any other class function, must be freed by calling `delete` in the class destructor.

5.2.2 Dynamic Storage Structures

A skilled programmer must understand how his declarations and instructions are mapped onto run-time objects. There are at least four different data structures that can all be called an "array of tickets". They have different runtime performance, different allocation syntax, and different deallocation requirements. The purpose of this section is to make the possibilities and the differences clear.

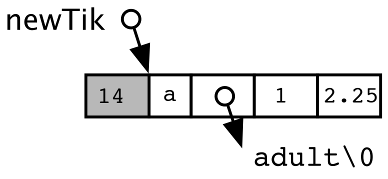
We begin with an image of the space allocated on the stack for a single ticket. The image shows the object created by the declaration `Ticket( 'a', 1, 2.25);`



A dynamically allocated object.
In both C and C++, when memory is dynamically allocated, the system must know how many bytes are in each dynamically allocated storage block, and that must be stored until the object is deallocated. The size can be calculated from the base type of the object, plus whatever padding is necessary to follow the compiler's byte-alignment rules. (Memory is allocated in blocks of size 2, 4, 8, etc.). The size can stored as part of the allocation block or kept in a hidden table, indexed by the first address of the block. The next diagram shows the system's view of the object created by the declaration

```
Ticket* newTik = new Ticket( 'a', 1, 2.25);
```

The programmer is not aware of the part with the gray background.

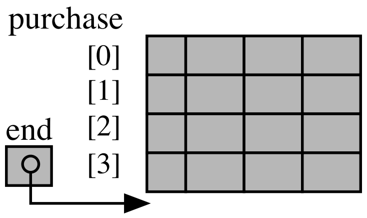


There is no rule in the language standard that the system must allocate a particular amount of overhead space, or where it must be. However, every compiler must do something of this sort. The diagram above illustrates an old and common strategy that was used for both C and C++ for many years. Extra space was allocated at the beginning of every dynamically allocated block and initialized to the actual number of bytes in the allocation block, including the overhead. In the diagram, this length field is shown against a pale gray background. Note that this field precedes the user's storage area, and the address returned by `new()` is the address of the beginning of the user's area.

In the following diagrams of arrays, we will continue to use Ticket objects, but omit the part names. These diagrams show the user's view of the storage; the allocation overhead will not be shown. Areas with a gray background are allocated on the stack. White areas are dynamically allocated.

A. A stack-allocated array of objects.
In this data structure, the size of everything is fixed at compile time and everything is created and freed automatically. It is created by declarations:

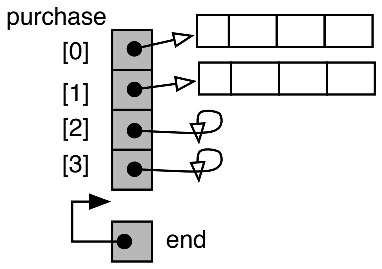
```
Ticket purchase[4];            // Requires a default Ticket constructor.
Ticket* end = purchase + 4;   // An off-board pointer to the end of the array.
```



If fixed-size stack allocation allows enough flexibility for the application, this one should be used. It is the most efficient in time and in space, and the simplest for memory management, since no calls on `new` and `delete` are needed.

B. A stack-allocated array of pointers to dynamic objects.
Here, the length of the array is determined at compile time, but the objects are created dynamically (usually in an input loop) and attached to the array during program execution . Some of the pointers might remain NULL. The declarations are:

```
Ticket* purchase[4];            // Does not require a default Ticket constructor.
Ticket** end = purchase + 4;   // An off-board pointer to the end of the array.
```



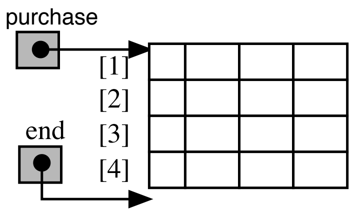
Again, the array portion of the data structure is allocated on the stack and its length is fixed. However, the objects are dynamically created and attached, so there is no need for a default object constructor, and it requires less total memory if the array never fills up. To free the memory for this data structure, loop around a delete statement:

```
for(int k=0, k<4, k++) delete purchase[k];
```

If the array is not full, this will call `delete` on a NULL pointer. That is OK.

C. A dynamic array of objects.
With this data structure, the length of the array is determined at run time and can be changed at runtime by attaching a different allocation area to the array name.

The stack-allocated part is a pointer to the array storage, which is allocated and initialized at runtime after the desired length is known.



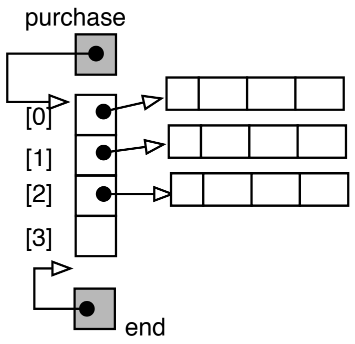
Memory management is needed but it is not difficult. The appropriate destructor is:

```
delete purchase;
```

This data structure is the basis of the C++ STL type `vector`. In `vector`, the initial allocation length is a parameter to the `vector` constructor or is a default. If there is a need for a longer array, a new one is allocated that is twice as long and the data is copied into it. This cycle of allocate - fill - double – allocate can be repeated as many times as needed[1].

The advantage of this kind of array is great runtime flexibility at a minimal performance penalty. The disadvantage is that, at runtime, the allocated array space will average only 3/4 full. If the objects and the number of objects are large, that can be a lot of unused memory.

D. A dynamically allocated array of pointers to dynamic objects.
In this data structure everything is dynamically allocated: both the array part and the objects[2]. Only the head pointer, and any scanning pointers, are on the stack.



Memory management is needed and requires two steps: first the loop from array version B, then the delete from version C. The appropriate destructor is:

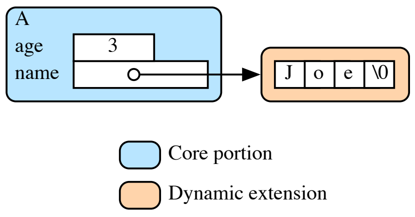
```
for(int k=0, k<4, k++) delete purchase[k];
delete purchase;
```

This fully dynamic data structure has maximum flexibility and does not require a program to allocate memory for objects it does not ever need. However, the savings in space come at the cost of added time: every access to one of the objects in the array requires an extra memory access to follow the pointer. An application that needs to be careful of execution time or the amount of energy used would be better off with the previous data structure.

5.3 Shallow Copy – Deep Delete

In C++, a distinction is made between *copying* one object into another and *moving* an object from one location to another. For simple objects, it makes no difference whether you copy or move. However, if an object has dynamically allocated data members, the difference is fundamentally important. The definitions and issues involved in copying and moving are presented in this section. The issues will be explained using objects of type `Mixed`, defined below. A Mixed object has a part that is dynamically allocated (orange) and a part (blue) that is not.

```
class Mixed {
    int age;
    char* name;
public:
    Mixed( int age, char* nm ) {
        int len = strlen( nm );
        name = new char[ len+1 ];
        strcpy( name, nm );
        this->age = age;
    }
    ~Mixed() { delete[] name; }
}
```

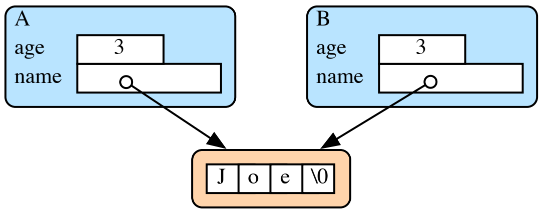


[1]Arrays that grow will be further discussed in this chapter and the next.
[2]This is how an array of objects is implemented in Java.

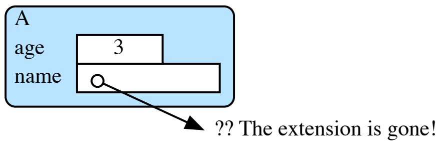
Copying. A *copy* of an object is made when the object is used to initialize a new object or it is passed as a call-by-value parameter to a function.

```
Mixed A = {3, "Joe"} // Create and initialize A.
Mixed B(A);          // Create B and initialize using copy constructor.
```

By default, every class is given a *copy constructor* so that these copies can be made. The default copy constructor performs a shallow copy, as pictured: core portions are copied, dynamic extensions are not. Integers, chars, and other primitive types are copied, as are pointers. However, the referent of a pointer is `not` copied.



After copying, the original object and the new object both point at the same dynamic extensions. This works perfectly well until a destructor gets called, then it fails disastrously because a properly written destructor frees the dynamic extensions of an object. For example, if B is a function parameter, and A was copied into it when the function was called, then both B and the shared dynamic extension will be deallocated when the function returns.



Copyable Types. A type is *copyable* if it can be copied without risking having its parts disappear when the copy is deallocated. A class that has only core parts is, by default, *copyable*.

Some very simple types, such as `char*` are not copyable. This is why C++ has the `string` class: `string` defines the set of functions necessary to make copies safely. A class with dynamic extensions is *not copyable* by default, but can be made copyable by defining or disabling a set of methods, including:

- The copy constructor (provided by default but may be redefined).
- Copy assignment (provided by default but may be redefined).
- A move constructor (no default definition).
- Move assignment (no default definition).

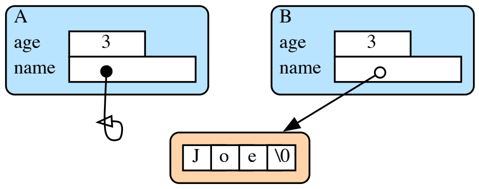
This is an advanced topic. For the curious, we will give examples of these at the end of this chapter. Most of the time, however, C++ programmers don't need to know about or define these methods. Programmers *do need* to follow some simple rules when using non-copyable class types:

Important warnings!

- Do not use call-by-value with these types; use call by reference or a pointer parameter to pass a non-copyable data object to a function. Here is the difference in the prototypes:

  Call-by-value: `myFunc ( Mixed m );` // The argument value will be copied into the parameter m.
  Call-by-reference: `myFunc ( Mixed& m );` // The name m will be an alias for the argument value.
  Call-by-reference: `myFunc ( const Mixed& m );` // Here. the argument value cannot be changed.
  Call-by-pointer: `myFunc ( Mixed* m );` // A pointer to the argument is stored in m.
  Call-by-pointer: `myFunc ( const Mixed* m );` // Here, the argument value cannot be changed.
- Use the stl string type, which is copyable, rather than using a char* as a member of a class.
- If your class is not copyable, and you need a container full of it, use a container of pointers to objects instead of a container of objects.
- When using stl container classes, check the documentation to find out whether they require the base type to be copyable.

Moving (not copying). Moving a non-copyable type is safe. To *move* an object means to transfer all its data to a new owner, and transfer responsibility for memory management to that owner also. If you move data from A to B, it means that A might no longer contain that data and must not be used to access it. The data now belongs to B. Moving data instead of copying it solves the problem of two objects being "joined at the hip" like conjoined twins.



To use move instead of copy, you need to define one or both of a move assignment and a move constructor. These methods should null-out any pointers in the origin object, making it safe to delete that object any time without affecting the new object.

5.4 Example: A Growing Array of Objects

In this section, we give the third version of the C++ Ticket program. It uses the same data class and controller class as V2, and adds a container class that defines a growing array. The main program and the Ticket class are identical to V2 and are not repeated here. The Vendor class has small variations and is given below The new class is `FlexArray`, which defines a growing array of objects and the functions to process the growing array.

FlexArray: the Container Class

This is a typical container class: it defines a data structure that can store many data objects and provides functions to make storage and retrieval easy. Like any good container class, it handles most of its own emergencies gracefully – including being too full. However, this is a simplified version of the FlexArray that works only for copyable classes. The method for putting data into the container is to copy it (not move it). At the end of the chapter we discuss the changes necessary to make it work with classes that have dynamic extensions.

A simple array has two limitations: both its base type and its length are fixed at the time the programmer writes the program. FlexArray removes these two limitations:

- Nothing in the FlexArray code depends on any aspect of the base type of the array, so it can be used to contain any type of objects. To reconfigure the FlexArray for a different bast type, the programmer must change a single `#include` and a single `typedef` at the top of the FlexArray header file, and recompile.
- The capacity of a FlexArray can increase during run time. The array inside the FlexArray is allocated at a default length. If it is full, and you try to put another object in it, the array will be reallocated to twice as large.

Growing. In order to be able to "grow" when needed, a FlexArray must have three data members:

- An array of base type BT.
- `n`, the number of BT objects currently stored in the array.
- `max`, the current allocation length of the array, which is the maximum number of BT's that can be stored in it.

When `n` equals `max`, the array is full. When the program then asks to put another BT into the array, it is an emergency! Rather than balk, and stop with an error, this class takes care of the emergency quietly. First, it allocates a new array, twice as long as the old one. Then it moves the data from the old array to the new array, and finally it frees the old array.

Why double the length each time? If you do the math and sum the series, you will see that, during the lifetime of the FlexArray, the total amount of time spent moving data is less than a constant factor times the current length of the array. In the usual notation for stating complexity, growing operates in linear time, or time = O(n). This is considered to be quite a reasonable cost for the benefits we get: programs that are more stable and do not crash when an unexpected quantity of data occurs.

The FlexArray header.

```
 1  //---------------------------------------------------------------------------
 2  //  flexarray.hpp
 3  //  TicketsV3a
 4  //
 5  //  Created by Alice Fischer on 7/15/15.
 6  //  Copyright (c) 2015 Alice Fischer. All rights reserved.
 7  //
 8  #ifndef FLEXARRAY
 9  #define FLEXARRAY
10
11  #include "tools.hpp"
12  #include "ticket.hpp"
13  typedef Ticket BT;
14
15  class FlexArray {
16    private:
17      int n;          // The number of data items currently stored in the array.
18      int max;        // The maximum number of data items the array can hold.
19      BT* purchase;   // A dynamically allocated array to store BT objects.
20
21      void grow();
22    public:
23      FlexArray( int len=4 );
24      ~FlexArray();
25      void push( BT& t );
26      BT& operator[]( int k );
27      int getN(){ return n; }
28  };
29  #endif //  FLEXARRAY
```

The typedef and matching #include.
This code is written in terms of an abstract type, BT (base type). To use the class, supply a typedef to make BT an alias for a real type. By using an abstract type with the typedef technique, we can configure the same container code to work with any base type. This will work with primitive types and programmer-defined classes.

- Line 12 includes the Ticket header so that Ticket is a known type when Line 13 is compiled.
- Line 13 configures the FlexArray to store Tickets; it defines that the abstract base type, BT, will be Ticket this time.

The class declaration.

- Lines 17–19 declare the three data members necessary to manage a growing array.
- Line 19 does not actually allocate an array– it simply declares a name for it. Allocation is done by the constructor.
- Line 21 is a private function that "grows" the array, when necessary. It is private because the array length is fully managed by this class.
- The public interface of this class consists of a constructor and ways to insert BT objects into the container and access them. It might be good to add a print function for debugging.
- Line 23: the constructor prototype. The default length for the array is 4 objects. A programmer can supply some other parameter, if he prefers.
- Line 24: a prototype for the destructor, which now has dynamic memory to free.
- Line 25: this function copies a BT object into the first open slot in the array.
- Line 26: this is an extension of the subscript operator. Iit will be discussed below.
- Line 27: a getter is supplied for the number of BT's in the array because this information is often needed.

FlexArray.cpp

```
30  //--------------------------------------------------------------------------
31  //  flexarray.cpp
32  //  TicketsV3a
33  //
34  //  Created by Alice Fischer on 7/15/15.
35  //  Copyright (c) 2015 Alice Fischer. All rights reserved.
36  //
37  #include "flexarray.hpp"
38
39  //--------------------------------------------------------------------------
40  FlexArray:: FlexArray( int length ) {
41      max = length;
42      n = 0;
43      purchase = new BT[ max ];
44  }
45
46  //--------------------------------------------------------------------------
47  FlexArray::~FlexArray() {
48      delete[] purchase;
49  }
50
51  //--------------------------------------------------------------------------
52  void
53  FlexArray:: grow(){
54      BT* temp = purchase;
55      purchase = new BT[ max*=2 ];
56      for (int k=0; k<n; ++k) purchase[k] = temp[k];
57      delete[] temp;
58  }
59
60  //--------------------------------------------------------------------------
61  void
62  FlexArray::push( BT& t ){
63      if (n == max) grow();
64      purchase[n++] = t;
65  }
66
67  //--------------------------------------------------------------------------
68  BT&
69  FlexArray::operator[]( int k ){
70      if (k>n) fatal("Walking off end of data.");
71      return purchase[k];
72  }
```

The constructor and destructor.

- The constructor allocates storage for an array of BT's and sets `n` and `max` to indicate that the container is empty, but could contain `length` objects.
- The destructor deletes the dynamic array that was allocated by the constructor. When deleting an array, use delete[].

The push function.

- FlexArray provides only one way to insert an object: `push()` inserts it at the end, just after the last existing object.
- Thus, the member `n` is both the number of objects in the array and the subscript of the first open space.
- If those two numbers are equal, the array is full and needs to be enlarged before doing the insertion. So `grow()` is called prior to actually inserting the object.
- The insertion (line 64) increments the counter. It is good practice to do these things in one statement.

The grow function. Grow() has three jobs: allocate new memory, move the data, free the old memory.

- Line 54 saves the pointer to the old memory.
- Line 55 doubles the `max` and uses the new value to allocate a new array.
- Line 56 copies the data from the old array to the new one. The BT must be copyable.
- Line 57 deletes the old array.

The subscript operator. C++ allows us to add new methods for any existing operator. This function defines a method subscript method for FlexArray, which is not an array. However, it composes an array. This method allows the programmer to subscript a FlexArray the same way as a simple array. It reaches inside the FlexArray to get the desired array element. In addition, it improves on the built-in subscript by adding a bounds check.

- Line 68: Subscript returns the address of the desired BT object.
- Line 69: the integer parameter is the subscript.
- Line 70: a subscript that is larger than `n` will cause a fatal error.
- Line 71: If all is well, the inner array is subscripted and the result is returned.

```
24  //---------------------------------------------------------------------------
25  //  vendor.hpp
26  //  TicketsV3a: version with a simple FlexArray.
27  //
28  //  Created by Alice Fischer on 7/13/15.
29  //  Copyright (c) 2015 Michael and Alice Fischer. All rights reserved.
30  //
31  #ifndef VENDOR
32  #define VENDOR
33
34  #include "tools.hpp"
35  #include "ticket.hpp"
36  #include "flexarray.hpp"
37
38  //---------------------------------------------------------------------------
39  class Vendor {
40    private:
41      //-----------------------------------------------------------------------
42      // This machine sells tickets at a fixed base price.  Tickets for children
43      //     are half price, and seniors are 80%.
44      const float adultPrice;
45      const float childPrice = round( 50 * adultPrice) / 100 ;
46      const float seniorPrice = round( 80 * adultPrice) / 100 ;
47      const string codes = "acs";  // Menu choice character codes
48      const string ages[3] = {"adult","child","senior"};
49
50      char ageCode;
51      int quantity;
52      size_t found;   // subscript of validated memu choice.
53
54      FlexArray purchase;
55      float totalPrice = 0;
56
57      // These functions are used only by other class functions.
58      void getOrder();
59      void instructions( ostream& out );
60
61    public:
62      Vendor( float basePrice ) : adultPrice( basePrice ){}
63      ~Vendor() =default;
64      void vend();
65      void print( ostream& out );
66  };
67  #endif // defined VENDOR
```

Changes in the Vendor class.

- The `#define` for the array length has been replaced by an `#include` for the FlexArray header.
- One of V2's eleven data members has changed (line 54): `purchase` is now a FlexArray of Tickets instead of a simple array of Tickets
- One of V2's eleven data members has disappeared. Since a FlexArray keeps track of how full it is, there is not need to redundantly store `nTiks`.
- The destructor has not changed, but this version uses the new syntax for memory-management methods. `=default` means that the programmer explicitly accepts the default definition provided by C++. This is exactly the same as writing `Vendor(){}`.

Notes on Vendor.cpp: `instructions()` , `getOrder()`, and `print()` have not changed since V2.

- Line 140 is the only change in the `vend()` function. V2 inserted the new ticket into an array. V3 pushes it into a FlexArray.

Vendor.cpp

```
 68  //------------------------------------------------------------------------------
 69  //  vendor.cpp
 70  //  TicketsV3a: version with a simple FlexArray.
 71  //
 72  //  Created by Alice Fischer on 7/13/15.
 73  //  Copyright (c) 2015 Michael and Alice Fischer. All rights reserved.
 74  //
 75
 76  #include "vendor.hpp"
 77  //------------------------------------------------------------------------------
 78  void Vendor :: instructions( ostream& out ) {
 79      out << "Please enter your purchases one at a time.  When you are finished "
 80          << "the total price will be charged to your charge card.\n";
 81  }
 82
 83  //------------------------------------------------------------------------------
 84  // Postcondition: the inputs are valid.
 85  void
 86  Vendor::getOrder() {
 87      char temp;      // to read in the age code. local.
 88
 89      cout <<fixed  <<setprecision(2)
 90          <<"\n    a: Adult:          $" << setw(5) << adultPrice
 91          <<"\n    c: Child under 12: $" << setw(5) << childPrice
 92          <<"\n    s: Senior Citizen: $" << setw(5) << seniorPrice <<"\n";
 93
 94      // Take an order for bus tickets, adult, child, or senior.
 95      for(;;){
 96          cout << "Please select a, c, or s: " ;
 97          cin >> temp;
 98          ageCode = tolower( temp );
 99          found = codes.find_first_of(ageCode);
100          if ( found != string::npos ) break;
101          cleanline(cin);   // Discard chars to end of line.
102          cout<<"Age selection must be a, c, or s; you entered: "<<ageCode <<endl;
103      }
104      for(;;){
105          cout << "Please enter the quantity: ";
106          cin >> quantity;
107          if (quantity>0 && quantity<10) break;
108          cleanline(cin);   // Discard chars to end of line.
109          cout <<"Quantity must be between 1 and 10 \n";
110      }
111  }
```

```

//----------------------------------------------------------------------------
void Vendor :: print( ostream& out ){
    out << fixed << setprecision(2)
        << "You bought the following tickets: \n";
    for ( int k=0; k<purchase.getN(); ++k )  {
        purchase[k].print( cout );
        cout << endl;
    }
    out << "Your credit card will be charged $" <<setw(6) << totalPrice <<endl;
}

//----------------------------------------------------------------------------
void Vendor :: vend(){
    float price;
    char answer = 'y';

    instructions( cout );
    do {
        getOrder();
        switch (ageCode) {
            case 'a': price = adultPrice; break;
            case 'c': price = childPrice; break;
            case 's': price = seniorPrice; break;
            default : price = 0;
        }
        Ticket tik( ages[found], quantity, price );
        totalPrice += tik.getPrice();
        purchase.push( tik );
        cout << "Do you want more tickets (y, n)?  ";
        cin >> answer;
    } while (tolower(answer) == 'y');
    cout << "Purchase has been concluded.\n";
}
```

The Ticket class has not changed since V2, and is not repeated here.

5.5 Defining Move Semantics

The simple version of FlexArray presented above relies on the fact that the BT is a copyable type. If BT has dynamic extensions, one would need to define move semantics for the base type or use a FlexArray of BT*s (data structure D in Section 2 of this chapter).

The Ticket class is not a convincing example of the need for move semantics, since it is copyable. In this section, we look at a very simple program that uses the Mixed class, (defined in Section 3) extended to define move semantics, and a main function that shows how to use move.

The Mixed class.
In this class, we explicitly define all seven of the related memory management operations. The parameters of the method determine what kind of constructor or assignment it is. The exact combinations of the class name, `const`, and `&` are critically important.

- A normal constructor, with parameters. This one allocates dynamic memory.
- A destructor that frees the dynamic memory.
- A default constructor so that we can create a blank object.
- The default copy constructor is retained as a private method so that the move constructor can use it but it cannot be used outside this class.
- The default copy assignment is also retained privately so that move assignment can use it but it cannot be used outside this class.

- The move constructor is defined to copy, then detach the dynamic memory from the original object.
- Move assignment is defined to copy, then detach the dynamic memory from the original object.

Note the keywords `=default` and `=delete` are useful both to be explicit about the intention of the code, and as a shorthand. For example, writing a full definition of how to copy an object would require one assignment statement per part of the object. Writing `=default` is brief.

```
 1  //---------------------------------------------------------------------------
 2  //  Mixed.hpp
 3  //  MixedMove
 4  //
 5  //  Created by Alice Fischer on 8/1/15.
 6  //  Copyright (c) 2015 Alice Fischer. All rights reserved.
 7  //
 8  #ifndef MIXED
 9  #define MIXED
10  #include "tools.hpp"
11
12  class Mixed {
13      int age;
14      char* name;
15      Mixed( const Mixed& t) = default;            // Private copy constructor.
16      Mixed& operator= (const Mixed& t) = default; // Private copy assignment
17
18  public:
19      ~Mixed() { delete[] name; }      // Define non-default destructor.
20
21      Mixed( int age, char* nm ) {     // Normal constructor with new.
22          this->age = age;
23          name = new char[ strlen( nm )+1 ];
24          strcpy( name, nm );
25      }
26      Mixed() =default;       // Use the null default constructor.
27
28      Mixed( Mixed&& t) {     // Move constructor.
29          *this = t;          // To begin, copy the whole object.
30          t.name = nullptr;   // Now null out string on the original
31      }
32
33      Mixed& operator= (Mixed&& t){         // move assignment
34          *this = t;          // To begin, copy the whole object.
35          t.name = nullptr;   // Now null out string on the original
36          return *this;       // this instance now owns the name.
37      }
38
39      void print(ostream& out){ out << age <<"\t" << name << endl; }
40  };
41  #endif //  MIXED
```

Using move semantics.

- This main program starts (line 55) by creating one Mixed object named `m1` using the normal constructor with parameters (line 21). This object has one dynamically allocated part: `name`. Unlike C++ strings, C-strings are not self managing. Therefore a destructor must be defined to delete the allocation.
- The Mixed destructor will be called every time a Mixed object *goes out of scope*. That happens when control leaves the block that created the object or leaves the function in which the object is a parameter.
- Line 56: we print m1 to verify that it was properly initialized.
- Line 68: we create another Mixed object named `m2` using the default constructor (line 26).
- Line 59 *moves* the object from `m1` to `m2` using *move assignment*. This move leaves `m1` undefined. Any further attempt, as on line 60, to use it or print it will cause a runtime error.

- We print `m2` on line 61 to verify that the data has been transferred to it.
- Line 63 uses the move constructor to declare and initialize `m3`. This leaves `m2` undefined, so the call on line 64 fails.
- We then print `m3` and see that it holds the data that was previously in `m2` and `m1`.
- The program ends cleanly. There are no problems with double deletion of the same thing.

```
42  //---------------------------------------------------------------------------
43  //  main.cpp
44  //  MixedMove
45  //
46  //  Created by Alice Fischer on 8/1/15.
47  //  Copyright (c) 2015 Alice Fischer. All rights reserved.
48  //
49  #include "tools.hpp"
50  #include "mixed.hpp"
51
52  int main( void ) {
53      cout << "Demonstrating move semantics. \n";
54      char buffer[] = "Luke";
55      Mixed m1 (6, buffer);
56      cout <<"m1 is:  "; m1.print( cout);
57
58      Mixed m2;
59      m2 = move(m1);              // Use move assignment.
60      //m1.print( cout);          // illegal memory access:  m1 is no more.
61      cout <<"m2 is:  "; m2.print( cout);
62
63      Mixed m3( move(m2) );       // Use move constructor.
64      //m2.print( cout);          // illegal memory access:  m2 is no more.
65      cout <<"m3 is:  "; m3.print( cout);
66
67      return 0;
68  }
```

Chapter 6: The STL Vector Class

6.1 The Standard Template Library–STL

A student of computer science must learn about data structures– these are classes that orgniize a collection of data for efficient storage and retrieval.

With any particular type of data structure, the operations performed on it depend wholly on the structure and not on the data stored in it. STL is a library of pre-programmed data structures written by the best C++ developers, in the form of templates.

A template is an abstract class definition. When a real type is supplied as a parameter, the template code is *instantiated* with that type to create a real class definition that can then be compiled. The result is code that is customized for the given type parameter. For example, STL provides a template for a stack class. Suppose your program defines an Item class. Then you would create a stack of Items, named `s`, like this:

```
stack<Item> s;
```

The STL library was designed with extreme care so that it is complete and portable and as safe as possible within the context of standard C++. Among the design goals were:

- To provide standardized and efficient template implementations of common data structures, and of algorithms that operate on these structures.
- To produce correct and efficient code.
- To unify array and linked list concepts, terminology, and interface syntax. This supports plug-and-play programming and permits a programmer to design and build much of an application before committing to a particular data structure.

There are three major kinds of components in the STL:

- Containers manage a set of storage objects (vector, list, stack, map, etc). Twelve basic kinds are defined, and each kind has a corresponding allocator that manages storage for it.
- Iterators are pointer-like objects that provide a way to traverse through a container.
- Algorithms (sort, set_union, make_heap, etc.) use iterators to act on the data in the containers. They are useful in a broad range of applications.

In addition to these components, STL has several kinds of objects that support containers and algorithms including key–value pairs, allocators (to support dynamic allocation and deallocation) and function-objects.

6.1.1 Containers

The C++ standard gives a complete definition of the functional properties and time/space requirements that characterize each container class. The intention is that a programmer will select a class based on the functions it supports and its performance characteristics. Although natural implementations of each container are suggested, the actual implementations are not standardized: any semantics that is operationally equivalent to the model code is permitted. Big-Oh notation is used to describe performance characteristics. In the following descriptions, an algorithm that is defined as time O(n), is no worse than O(n) but may be better.

Member operations. Some member functions are defined for all containers. These include: Constructors a destructor, traversal initialization: (begin(), end()), size() (current fill level), max_size())current allocation size), and empty() (true or false).

Other functions are defined only for a subset of the containers, or for a particular container. For more information, go to `cplusplus.com` and click on Reference, then Containers, then <vector> or the name of another container.

6.2 Using the STL vector Class

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

//--------------------------------------------------------------------
void print(const vector<int>& vec)  // print the elements of the vector
{
    int count = vec.size();
    for (int idx = 0; idx < count; idx++)
        cout << "Element " << idx << " = " << v.at(idx) << endl;
    cout << "--- done ---\n";
}

//--------------------------------------------------------------------
int main( void )
{
    vector<int> ages;               // create a vector of int's

    // insert some numbers in random order
    ages.push_back(11);      ages.push_back(82);      ages.push_back(24);
    ages.push_back(56);      ages.push_back(6);

    cout << "Before sorting: " <<endl;
    print(ages);                            // print vector elements

    // sort vector elements
    sort(ages.begin(), ages.end());
    cout << "\nAfter sorting: " <<endl;
    print(ages);                            // print elements again

    // search the vector for the number 3
    int val = 3;
    vector<int>::iterator pos;
    pos = find(ages.begin(), ages.end(), val);
    if (pos == ages.end())
        cout << "\nThe value " << val << " was not found" << endl;

    // print the first element
    cout << "First element in vector is " << ages.front() << endl;

    // remove last element
    cout << "\nNow remove last element and element=24 "<< endl;
    ages.pop_back();

    // remove an element from the middle
    val = 24;
    pos = find(ages.begin(), ages.end(), val);
    if (pos != ages.end())  ages.erase(pos);

    // print vector elements
    print(ages);
    return 0;
}
```

The output: (condensed into three columns. Read left column first.)

```
Before sorting:             After sorting:
Element 0 = 11              Element 0 = 6
Element 1 = 82              Element 1 = 11
Element 2 = 24              Element 2 = 24
Element 3 = 56              Element 3 = 56
Element 4 = 6               Element 4 = 82
--- done ---                --- done ---
The value 3 was not found
First element in vector is 6

Now remove last element and element=24
Element 0 = 6
Element 1 = 11
Element 2 = 56
--- done ---
```

In this example program we create a vector of ints and an iterator for it, populate the vector, sort it, search it, remove some items and print it. The STL vector functions used are the null constructor, `push_back(), pop_back(), at(), size(), sort(), find(), erase(), front(), begin(), and end().` The vector class also supports the subscript operator, `insert()`, and many other functions.

- Both FlexArray and vector copy the data values into the data structure. This means that the base type must be copyable.
- Both of these data structures grow. However, vector also supports iterators. It is essential to remember that any iterators in use are invalidated if the data structure grows or if elements are removed from the data structure.

Notes on the vector example:

- Line 2: the header file needed for vectors.
- Line 3: the header file needed for —tt sort().
- Line 7: a vector parameter. Notice the use of `&`: that indicated call-by-reference and means that the parameter will be the address of the vector, not the vector itself. It is important to NOT copy large data structures.
- Line 7: the type qualifier `const` means that the print function will not modify the vector.
- Line 9: get the number of ints currently stored in the vector.
- Line 11: the function `at()` calls subscript on the array that is inside the vector.
- An ordinary `for` loop was used in this example to illustrate calls on `size()` and `at()`. However, it is not the easiest or most modern way to write the print loop. Here is the modern alternative, using a for-each loop:

```
void print(const vector<int>& vec) // print the elements of the vector
{
    for (int value : vec) cout << value << endl;
    cout << "--- done ---\n";
}
```

- Line 18: we construct a vector, given the type of element to store within it. Initially this vector is empty.
- Lines 21–22: we put five elements into the vector (each goes at the end).
- Line 25 and 30: we print the vector before and after sorting the data.
- Line 28: `begin()` and `end()` are functions that are defined on all containers. They return iterators associated with the first and last elements in the container. (`end()` is actually a pointer to the first array slot past the end of the vector.)
- Line 28: sort is one of the algorithms supported by vector. The arguments to sort are two iterators: one for the beginning and the other for the end of the portion of the vector to be sorted.
- Line 34: we declare an iterator variable of the right kind for vector¡int¿.
- Lines 35 and 48: the `find()` function searches part of the vector (specified by two iterators) for a key value (the third argument). On these lines we search the vector for the data values 3 and 24, and store the result in the iterator.

- Lines 36 and 49 test for the value `end()`, which is returned by the find function to signal failure to find the key value.
- Line 40: get the first element in the vector but do not erase it from the vector.
- Line 44: remove the last element from the vector: very efficient.
- Line 49: remove an element from the middle of a vector, using an iterator: not as efficient as `pop_back()`.

Warnings: A STL vector is a generalization of a FlexArray. You might want to use vector it because it is not as restricted as FlexArray and it is a standard type that presents the same interface as the other STL sequence container classes. The FlexArray, however, is simpler and easier to use for those things it does implement.

6.3 Using the STL string Class

This brief string demo uses only a few of the string functions: one constructor, string assignment, string comparison, `length(), size(), c_str(), find(), find_first_of(), find_last_of(), substr(), and replace()`.

```
#include <string>                             // Header file for STL strings
#include <iostream>
using namespace std;

int main( void )
{
    // Allocate, initialize, measure, and print.
    string str0;            // Use the null constructor.
    string str1("This is string number one.");
    cout << "\nString str1 is: \"" << str1.c_str() <<"\". "
        <<" Its length is: " << str1.size() << "\n\n";

    // search second string1 for first instance of the letter 'x'.
    size_t idx = str1.find_first_of("x");
    if (idx != string::npos)
        cout << "The first 's' in string str2 is at pos " << idx <<"\n\n";

    // search string1 for last instance of the letter 'e'
    idx = str1.find_last_of("e");
    if (idx != string::npos)
        cout << "The last 'e' in string str1 is at pos " << idx <<endl;
    else cout << "No char 'e' found in string str1" <<endl;

    // search second string1 for first instance of the letter 'x'.
    idx = str1.find_first_of("x");
    if (idx != string::npos)
        cout << "The first 'x' in string str2 is at pos " << idx <<"\n\n";
    else cout << "No char 'x' found in string str2\n\n";

    // substring and string assignment
    cout << "Get a substring of six letters starting at subscript 8: ";
    string str2 = str1.substr(8,6);
    cout << str2.c_str() << endl;

    // string comparison
    cout << "Compare two strings for equality using == : ";
    if (str2 == "string") cout <<"it is easy to compare two strings.\n\n";
    cout << str2.c_str() << endl;

    cout << "Now replace \"string\" with \"xxxyyyxxx\".\n";
    idx = str1.find("string");
    if (idx != string::npos)
        str1.replace(idx, string("string").length(), "xxxyyyxxx");
    cout << "str1 with replacement is: " << str1  << "\n\n";
    return 0;
}
```

The output:

```
String str1 is: "This is string number one.".  Its length is: 26

The last 'e' in string str1 is at pos 24
No char 'x' found in string str2

Get a substring of six letters starting at subscript 8: string
Compare two strings for equality using == : it is easy to compare two strings.

string
Now replace "string" with "xxxyyyxxx".
str1 with replacement is: This is xxxyyyxxx number one.
```

Notes on the string example. This example uses the following string functions:

- A string is a vector of chars. This gives a string the ability to grow, as needed, to contain any input.
- Line 1: include the header file needed for this class.
- Line 8: construct a C++ string using the null constructor. The result is an empty string with length 0.
- Line 9: construct a C++ string from a C string.
- Line 10: how to get a C-style string out of a C++ string.
- Line 11: `size()` returns the length of the string. It is a synonym for `string::length()`.
- Lines 14 and 25: use `find_first_of()` to locate a char in a string.
- Related methods, `find_last_of()` and `find()` are shown on lines 19 and 41.
- Line 19: you can search for either the first or the last occurrence of a letter. If you want to find all occurrences, you can supply a second parameter, which is the position (type size_t) at which you want to start searching.
- The `find()` methods return `str::npos` when the search fails.
- Line 32: creating a new string from a substring of another.
- Line 43 calls the `length()` function, which is a synonym for `size()`.
- Line 43: replace a substring with another string. Note that the old and new substrings do not need to be the same length.

The string library also supports the subscript operator, `at()` (another way to do a subscript), `append`, >>, <<, `+` (concatenate) and `getline()`. For more information go to `cplusplus.com` and click on Reference, then Other, then <string>.

6.3.1 Algorithms.

The library of stl algorithms contains over 80 different functions. It is difficult to characterize them or even to describe the range of things they do. Instead, we will describe a few of the most useful"

- `binary_search( begin_range, end_range, value)`:
  Returns true if value is found in the container, false otherwise.
- `copy(begin_source, end_source, destination)`:
  Copies a range of values into the destination container.
- `count(first, last,, value)`:
  Returns the number of times the value occurs in the range.
- `fill(first, last,, value)`:
  Fills the specified range with copies of the value.
- `find(first, last, value)`:
  Returns an iterator to the first element in the range that equals val. Otherwise, returns last.
- `is_sorted(first, last)`:
  true if the range [first,last) is sorted into ascending order, false otherwise.

- `lexicographical_compare()`: Compare for alphabetical order
- `min()` and `max()`: Return the smaller (or larger) of two values.
- `reverse(first, last)`:
  Reverses the order of the elements in the range Calls iter_swap to swap elements.
- `shuffle(first, last)`:
  Rearranges the elements in the range randomly, using a uniform random number generator.
- `sort(first, last)`:
  Sorts the elements in the range into ascending order.
- `stable_sort(first, last)`:
  Like sort, but stable_sort preserves the relative order of the elements with equivalent values.
- `swap()`:
  Exchange the values of two same-type objects.
- `unique(first, last)`:
  Removes all but the first element from every consecutive group of equivalent elements in the range

6.4 Example: Using a Vector

In this section, we give the fourth version of the C++ Ticket program. It uses the same data class and controller class as V2, but replaces the FlexArray with a built-in data structure, `vector`. Vector is a much more extensive and sophisticated class than `FlexArray`, but it works in the same way: by doubling, as needed.

Vector is also a generic class; it achieves generality by being defined as a class template. This, also, is much the same as FlexArray, but the syntax and compiler mechanism are more modern and more powerful than the typedef technique.

Vector is probably the simplest stl container, and certainly the most useful. So it makes a good place to start learning how to use stl classes.

Most of version 4 of the Ticket program is exactly like version 3. Only the changed parts are repeated here, with the same line numbers as in version 3 to make comparison easier.

Tickets, Version 4.

- main.cpp : identical
- ticket.hpp : identical
- ticket.hpp : identical
- flexarray.hpp : gone, replaced by stl vector.
- flexarray.cpp : gone.
- vendor.hpp : line 53 now instantiates a vector instead of a FlexArray.
- vendor.cpp : the for loop in the print function is different, as is the push command on line 141.

vendor.hpp.

- In the vendor header file, we need to include vector.h instead of flexarray.hpp. Since tools.hpp already includes vector.h, there is simply one fewer `#include`.
- Line 53 is the only difference in the class declaration: it instantiates a vector instead of a FlexArray.
- Vector is a template class – it is abstract until you provide a type parameter. To instantiate a template class, you must supply the angle brackets and, within them, the base type of the data structure. Any previously defined type is acceptable, it does not need to be a class type.

```
#ifndef VENDOR
#define VENDOR

#include "tools.hpp"
#include "ticket.hpp"


//-------------------------------------------------------------------------------
class Vendor {
  private:
    //-----------------------------------------------------------------------------
    // This machine sells tickets at a fixed base price.  Tickets for children
    //     are half price, and seniors are 80%.
    const float adultPrice;
    const float childPrice = round( 50 * adultPrice) / 100 ;
    const float seniorPrice = round( 80 * adultPrice) / 100 ;
    const string codes = "acs";  // Menu choice character codes
    const string ages[3] = {"adult","child","senior"};

    char ageCode;
    int quantity;
    size_t found;   // subscript of validated memu choice.

    vector<Ticket> purchase;
    float totalPrice = 0;

    // These functions are used only by other class functions.
    void getOrder();
    void instructions( ostream& out );

  public:
    Vendor( float basePrice ) : adultPrice( basePrice ){}
    ~Vendor() =default;
    void vend();
    void print( ostream& out );
};
#endif // defined VENDOR
```

print.cpp.
The vector class supports use of the for-each loop, as illustrated here.

- A for-each loop defines an `iterator`, which is an stl type that works like a pointer. The iterator is used to walk through the container and point at each object stored there, one at a time, in order.
- In this loop (line 118) the local name `t` refers to the current element in the vector. You read this: " for each Ticket, t, in purchase, do this block of code."
- Note that `t` is a reference to the object, not the subscript of the object.
  Write `t.print(cout);` not `purchase[t].print(cout);`

```
114  //-------------------------------------------------------------------------------
115  void Vendor :: print( ostream& out ){
116      out << fixed << setprecision(2)
117          << "You bought the following tickets: \n";
118      for (Ticket t : purchase)  {
119          t.print( cout );
120          cout << endl;
121      }
122      out << "Your credit card will be charged $" <<setw(6) << totalPrice <<endl;
123  }
```

vend.cpp.
Line 141 inserts the new ticket into the vector; `push_back` works the same way as FlexArray::push(). Both copy (not move) the object into the internal array. That means the objects must be copyable[1].

```
125  //----------------------------------------------------------------------------
126  void Vendor :: vend(){
127      float price;
128      char answer = 'y';
129
130      instructions( cout );
131      do {
132          getOrder();
133          switch (ageCode) {
134              case 'a': price = adultPrice; break;
135              case 'c': price = childPrice; break;
136              case 's': price = seniorPrice; break;
137              default : price = 0;
138          }
139          Ticket tik( ages[found], quantity, price );
140          totalPrice += tik.getPrice();
141          purchase.push_back( tik );
142          cout << "Do you want more tickets (y, n)?  ";
143          cin >> answer;
144      } while (tolower(answer) == 'y');
145      cout << "Purchase has been concluded.\n";
146  }
```

[1]Either the base class must have no dynamic extensions or move semantics must be defined for the base class.