Applied C and C++ Programming

Alice E. Fischer
David W. Eggert
University of New Haven

Michael J. Fischer
Yale University

August 2018

Part IV

Representing Data

Chapter 12

Strings

In this chapter, we begin to combine the previously studied data types into more complex units: strings and arrays of strings. Strings combine the features of arrays, characters, and pointers discussed in the preceding chapters. String input and output are presented, along with the most important functions from the standard —pl C++`string` library. Then strings and arrays are combined to form a data structure called a *ragged array*, which is useful in many contexts. Examples given here include a two applications with menu-driven user interfaces.

Types and type names. Variable names are like nouns in English; they are used to refer to objects. Type names are like adjectives; they are used to describe objects but are not objects themselves. A type is a pattern or rule for constructing future variables, and it tells us how those variables may be used. Types such as `int`, `float`, and `double` have names defined by the `C` standard. These are all simple types, meaning that objects of these types have only one part. We have also studied arrays, which are aggregate objects.

Other important types that we use all the time, such as cstring, do not have standard names in `C`; they must be defined by type specifications[1]. For example, although the `C` standard refers many times to strings, it does not define `string` as a type name; the actual type of a string in `C` is `char*`, or "pointer to character."

12.1 String Representation

We have used literal strings as formats and output messages in virtually every program presented so far but have never formally defined a string type. We will use the typename *cstring* to refer to `C`-style strings. This is a necessary type and fundamental. `C` reference manuals talk about strings and the standard `C` libraries provide good support for working with strings. The functions in the `stdio` library make it possible to read and write strings in simple ways, and the functions in the `Cstring` library let you manipulate strings easily.

In contrast, `C++` defines a type `string` and a `string` library of functions that are similar but not identical to the functions in the `Cstring` library. In this text, we will focus on the `C++` strings and library. However, the C++ programmer must also understand `C` strings because they are still part of the language and are used for a variety of essential purposes.

Within `C`, the implementation of a **string** is not simple; it is defined as a pointer to a null-terminated array of characters. The goal in this section is to learn certain fundamental ways to use and manipulate strings and to learn enough about their representation to avoid making common errors. To streamline the coding, the following type definition will be used to define the type `cstring` as a synonym for a `const char*`

The `C++` string is a much more sophisticated type and is far easier to use. A `C++string` has a `Cstring` inside of it, plus additional information used for managing storage[2]. Happily, `C++strings` can be used without understanding how or why they work.

[1] In contrast, String is a standard type in Java and `string` is standard in C++.

[2] This will be covered in Chapter 16.

A simple string literal:	`"George Washington\n"`
A two-part string literal:	`"was known for his unswerving honesty "`
	`"and became our first president."`
One with escape codes:	`"He said \"I cannot tell a lie\"."`

Figure 12.1. String literals.

Suppose we want a string that contains different messages at different times. There are three ways to do this! First, it is possible to simply have an array of characters, each of which can be changed individually to form a new message. Second, the pointer portion of the `cstring` can be switched from one literal array of letters to another. Third, you can define a `C++ string` that can contain any sequence of letters, of any length. It will enlarge itself to store whatever you put in it. Also, the individual letters in a `C++ string` can be searched and changed, if needed.

These three string representations have very different properties and different applications, leading directly to programming choices, so we examine them more closely.

12.1.1 String Literals

A **string literal** is a series of characters enclosed in double quotes. String literals are used for output, as formats for `printf()` and `scanf()`, and as initializers for both `C strings` and `C++ strings`. Ordinary letters and escape code characters such as `\n` may be written between the quotes, as shown in the first line of Figure 12.1. In addition to the characters between the quotes, the C translator adds a null character, `\0`, to the end of every string literal when it is compiled into a program. Therefore, the literal `"cat"` actually occupies 4 bytes of memory and the literal `"$"` occupies 2.

Two-part literals. Often, prompts and formats are too long to fit on one line of code. In old versions of C, this constrained the way programs could be written. The ANSI C standard introduced the idea of *string merging* to fix this problem. With **string merging**, a long character string can be broken into two or more parts, each enclosed in quotes. These parts can be on the same line of code or on different lines, so long as there is nothing but whitespace or a comment between the closing quote of one part and the opening quote of the next. The compiler will join the parts into a single string with one null character at the end. An example of a two-line string literal is shown in the second part of Figure 12.1.

Quotation marks. You can even put a quote mark inside a string. To put a single quotation mark in a string, you can write it like any other character. To insert double quotes, you must write `\"`. Examples of quotes within a quoted string are shown in the last line of Figure 12.1.

Strings and comments. The relationship between comments and quoted strings can be puzzling. You can put a quoted phrase inside a comment and you can put something that looks like a comment inside a string. How can you tell which situation is which? The translator reads the lines of your source code from top to bottom and left to right. The symbol that occurs first determines whether the following code is interpreted as a comment or a string. If a begin-comment mark is found after an open quote, the comment mark and the following text become part of the string. On the other hand, if the `//` or `/*` occurs first, followed by an open quote, the following string becomes part of the comment. In either case, if you attempt to put the closing marks out of order, you will generate a compiler error. Examples of these cases are shown in Figure 12.2.

12.1.2 A String Is a Pointer to an Array

We have said that a cstring is a series of characters enclosed in double quotes and that a cstring is a pointer to an array of characters that ends with the null character. Actually, both these seemingly contradictory

A comment in a string:	`"This // is not a comment it is a string."`
A string in a comment:	`// Here's how to put "a quote" in a comment.`
Overlap error:	`"Here /* are some illegal" misnested delimiters. */`
Overlap error:	`/* We can't "misnest things */ this way, either."`

Figure 12.2. Quotes and comments.

statements are correct: When we write a string in a program using double quotes, it is translated by the compiler into a pointer to an array of characters.

String pointers. One way to create a string in C is to declare a variable of type `cstring` or `char*` and initialize it to point at a string literal.[3] Figure 12.4 illustrates a **cstring variable** named `word`, which contains the address of the first character in the sequence of letters that makes up the literal `"Holiday"`. The pointer variable and the letter sequence are stored in different parts of memory. The pointer is in the user's memory area and can be changed, as needed, to point to a different message. In contrast, the memory area for literals cannot be altered. C and C++ forbid us to change the letters of `"Holiday"` to make it into `"Ponyday"` or `"Holyman"`.

Null objects. The null character, the null pointer, and the null string (pictured in Figure 12.3) often are confused with each other, even though they are distinct objects and used in different situations.

The **null string** is a string with no characters in it except the null character. It is denoted in a program by two adjacent double quote marks, `""`. It is a legal string and often useful as a placeholder or initializer. Both the null pointer and the null string are pointers, but the first is all 0 bits, the address of machine location 0, and the second is the nonzero address of a byte containing all 0 bits.

The **null character** (eight 0 bits), or **null terminator** is used to mark the end of every string and plays a central role in all string processing. When you read a cstring with `scanf()`, the null character automatically is appended to it for you. When you print a cstring with `printf()` or `puts()`, the null character marks where to stop printing. The functions in the C and C++ `string` libraries all use the null character to terminate their loops. See Figure 12.3.

[3] Dynamically allocated memory also can be used as an initializer. This advanced topic is left for Chapter 16.

- The null character is 1 byte (type `char`) filled with 0 bits.
- The null pointer is a pointer filled with 0 bits (it points to address 0). In C it is called `NULL`. In C++ it is called nullptr.
- The null string is a pointer to a null character. The address in the pointer is the nonzero address of some byte filled with 0 bits.

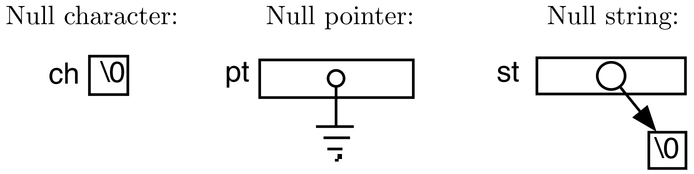


Figure 12.3. Three kinds of nothing.

```
typedef char* cstring;
cstring word = "Holiday";
```

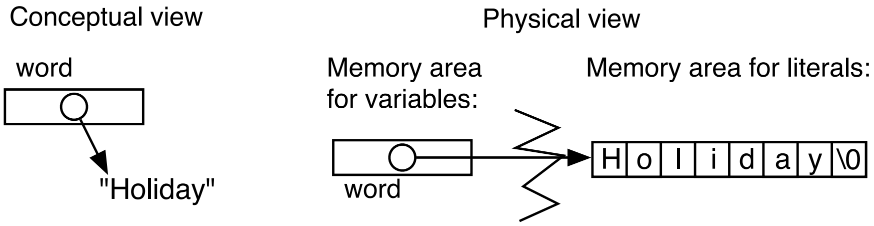


Figure 12.4. Implementation of a cstring variable.

Using a pointer to get a string variable. We can use a `char*` variable and **string assignment** to select one of a set of predefined messages and remember it for later processing and output. The `char*` variable acts like a finger that can be switched back and forth among a set of possible literals.

For example, suppose you are writing a simple drill-and-practice program and the correct answer to a question, an integer, is stored in `target_answer` (an integer variable). Assume that the user's input has been read into `input_answer`. The output is supposed to be either the message "`Good job!`" when a correct answer is entered or "`Sorry!`" for a wrong answer. Let `response` be a string variable, as shown in Figure 12.5. We can select and remember the correct message by using an `if` statement and two pointer assignment statements, as shown.

12.1.3 Declare an Array to Get a String

To create a cstring whose individual letters can be changed, as when a word is entered from the keyboard, we must create a character array big enough to store the longest possible word. Figure 12.6 shows one way to do this with an array declaration.[4] Here, the variable `president` is a character array that can hold up to 17 letters and a null terminator, although only part of this space is used in this example.

Initializers. A character array can be declared with or without an initializer. This initializer can be either a quoted string or a series of character literals, separated by commas, enclosed in braces. The array `president` in Figure 12.6 is initialized with a quoted string. The initializer also could be

```
 = {'A','b','r','a','h','a','m',' ', 'L','i','n','c','o','l','n','\0'}
```

[4] The use of dynamic allocation for this task will be covered in Chapter 16.

```
if (input_answer == target_answer) response = "Good job!";
else response = "Sorry!";
```

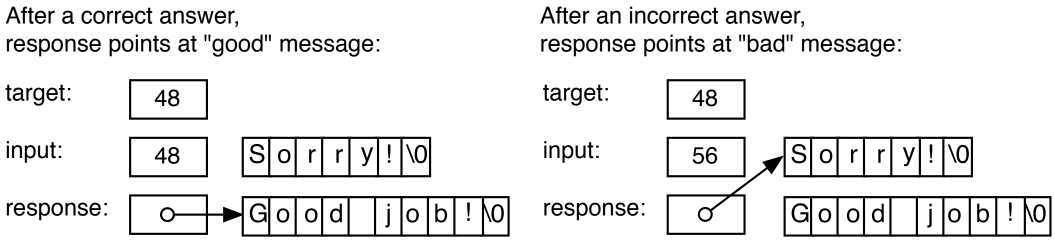


Figure 12.5. Selecting the appropriate answer.

An array of 18 `char`s containing the letters of a 15-character string and a null terminator. Two array slots remain empty.

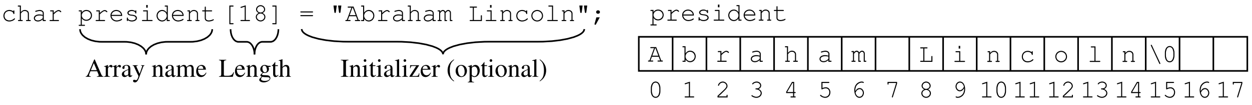


Figure 12.6. An array of characters.

This kind of initializer is tedious and error prone to write, and you must remember to include the null character at the end of it. Because they are much more convenient to read and write, quoted strings (rather than lists of literal letters) usually are used to initialize `char` arrays. The programmer must remember, though, that every string ends in a null character, and there must be extra space in the array for the `\0`.

Using character arrays. If you define a variable to be an array of characters, you may process those characters using subscripts like any other array. Unlike the memory for a literal, this memory can be updated.

It also is true that an array name, when used in C with no subscripts, is translated as a pointer to the first slot of the array. Although an array is not the same as a string, this representation allows the name of a character array to be *used* as a string (a `char*`). Because of this peculiarity in the semantics of C, operations defined for strings also work with character arrays. Therefore, in Figure 12.6, we could write `puts( president )` and see `Abraham Lincoln` displayed.

However, an array that does not contain a null terminator must not be used in place of a string. The functions in the `string` library would accept it as a valid argument; the system would not identify a type or syntax error. But it would be a semantic error; the result is meaningless because C will know where the string starts but not where it ends. Everything in the memory locations following the string will be taken as part of it until a byte is encountered that happens to contain a 0.

12.1.4 Array vs. String

Typical uses of strings are summarized in Figure 12.7. Much (but not all) of the time, you can use strings without worrying about pointers, addresses, arrays, and null characters. But often confusion develops about the difference between a `char*` and an array of characters. A common error is to declare a `char*` variable and expect it to be able to store an array of characters. However, a `char*` variable is only a pointer, which usually is just 4 bytes long. It cannot possibly hold an entire word like `"Hello"`. If we want a string to store alphabetic data, we need to allocate memory for an array of characters. These issues can all be addressed by understanding how strings are implemented in C.

Figure 12.8 shows declarations and diagrams for a character array and a `char*` variable, both initialized by quoted strings. As you can see, the way storage is allocated for the two is quite different. When you use

Use type `char*` for these purposes:

- To point at one of a set of literal strings.
- As the type of a function parameter where the argument is either a `char*` or a `char` array.

Use a `char` array for these purposes:

- To hold keyboard input.
- Any time you wish to change some of the letters individually in the string.

Figure 12.7. Array vs. string.

These declarations create and initialize the objects diagrammed below:

```
char ara[7] = "Monday";   // 7 bytes total.
char* str   = "Sunday";   // 11 bytes total, 4 four pointer, 7 for chars.
char* str2   = &str[3];   // 4 more bytes for the pointer.
```

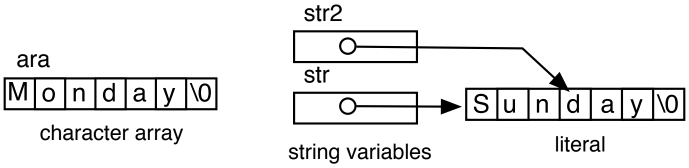


Figure 12.8. A character array and a string.

an array as a container for a string, space for the array is allocated with your other variables, and the letters of an initializing string (including the null) are copied into the array starting at slot 0. Any unused slots at the end of the array are left untouched and therefore contain garbage. In contrast, two separate storage areas are used when you use a string literal to initialize a pointer variable such as `str`. A string pointer typically occupies 4 bytes while the information itself occupies 1 byte for each character, plus another for the null character. The compiler stores the characters and the null terminator in an unchangeable storage area separate from the declared variables, and then it stores the address of the first character in the pointer variable.

12.1.5 C++ Strings

A C++ string contains a C string plus memory management information. The array part of the string is dynamically allocated, and the length of the allocation is stored as part of the object. (This is labelled "max" in Figure 12.9) The other integer, labelled "len", is the actual length of the string.

When a variable is created and initialized to a string value, enough space is allocated to store the value plus a null terminator, plus padding bytes to make the length of the entire allocation a power of 2 (8, 16, 32 bytes, etc.)[5]. If the string becomes full and more space is needed, the initial space is doubled. In this way, a C++ string variable can hold a string of any length: it "grows" when needed to hold more characters.

Because of the automatic resizing, C++ strings are strongly preferred for all situations in which the length of the string is unknown or not limited.

[5] This is a simplified explanation. Modern compilers also have optimized representations for short strings.

These declarations create the C++ string objects diagrammed below.

```
string s1 = "Monday";
string s2("Tuesday");
```

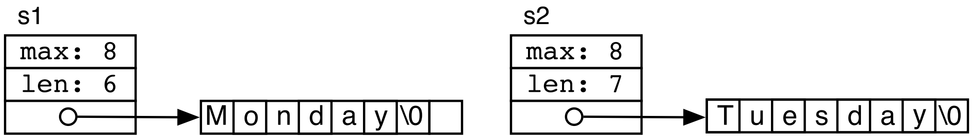


Figure 12.9. C++ strings.

```
#include <stdio.h>

int main( void )
{
    char* s1 = "String demo program.";
    char* s2 = "Print this string and ring the bell.\n";

    puts( s1 );
    printf( "\t This prints single ’ and double \" quotes.\n" );
    printf( "\t %s\n%c", s2, ’\a’ );

    puts( "These two quoted sections " "form a single string.\n" );
    printf( "You can break a format string into pieces if you\n"
            "end each line and begin the next with quotes.\n"
            "You may indent the lines any way you wish.\n\n" );

    puts( "This\tstring\thas\ttab\tcharacters\tbetween\twords.\t" );
    puts( "Each word on the line above starts at a tab stop. \n" );
    puts( "This puts()\n\t will make 3 lines of output,\n\t indented.\n" );

    printf( " >>%-35s<< \n >>%35s<< \n",
            "left justify, -field width", "right justify, +field width" );
    printf( " >>%10s<< \n\n", "not enough room?  Field gets expanded" );
}
```

Figure 12.10. String literals and string output in C.

12.2 C String I/O

Even though the string data type is not built into the C language, functions in the standard libraries perform input, output, and other useful operations with strings. Several code examples are given in this chapter. Because it is important to know how to do input and output in both languages, these examples are given first in C, then in C++.

12.2.1 String Output

We have seen two ways to output a string literal in C: `puts()` and `printf()`. The `puts()` function is called with one string argument, which can be either a literal string, the name of a string variable, or the name of a character array. It prints the characters of its argument up to, but not including, the null terminator, then *adds* a newline (`\n`) character at the end. If the string already ends in `\n`, this effectively prints a blank line following the text. We can call `printf()` in the same way, with one string argument, and the result will be the same except that a newline will not be added at the end. The first box in Figure 12.10 demonstrates calls on both these functions with single arguments.

String output with a format. We can also use `printf()` with a `%s` conversion specifier in its format to write a string. The string still must have a null terminator. Again, the corresponding string argument may be a literal, string variable, or `char` array. Examples include

```
string fname = "Henry";
printf( "First name: %s\n", fname );
printf( "Last name: %s\n", "James" );
```

```
// ---------------------------------------------------------------------
//  Figure 12.11:  String literals and string output in C++.
//  Rewritten in C++ June 21, 2016
// ---------------------------------------------------------------------
#include "tools.hpp"
typedef const char* cstring;

int main( void )
{
    cstring s1 = "\nString demo program.";
    cstring s2 = "Print this string and a newline.\n";

    cout << s1 <<endl;
    cout << s2;
    cout << "\t This line prints single ’ and double \" quote marks.\n\n" ;

    cout << "These two quoted sections " "form a single string literal.\n" ;
    cout << "You can break a format string onto several lines if you end \n"
            "each line with a quote and begin the next line with a quote.\n"
            "You may indent the lines any way you wish.\n\n";

    cout << "This\tstring\thas\ta\ttab\tcharacter\tafter\teach\tword.\n";
    cout << "Each word on the line above should start at a tab stop. \n";
    cout << "This line\n \t will make 3 lines of output,\n \t indented.\n\n";

    cout <<"[[" <<left <<setw(35) <<"A left-justified string" <<"]]\n";
    cout <<"[[" <<right <<setw(35) <<"and a right-justified string" <<"]]\n";
    cout <<"[[" <<left <<setw(35) <<"Not enough room?  Field gets expanded." <<"]]\n";
    cout <<endl;
}

/* Output: ------------------------------------------------------------

String demo program.
Print this string and a newline.
     This line prints single ’ and double " quote marks.

These two quoted sections form a single string literal.
You can break a format string onto several lines if you end
each line with a quote and begin the next line with a quote.
You may indent the lines any way you wish.

This    string  has a   tab  character  after   each    word.
Each word on the line above should start at a tab stop.
This line
     will make 3 lines of output,
     indented.

[[A left-justified string            ]]
[[       and a right-justified string]]
[[Not enough room?  Field gets expanded.]]

*/
```

Figure 12.11. String literals and string output in C++.

The output from these statements is:

```
First name: Henry
Last name:  James
```

Further formatting is possible. A `%`*n*`s` field specifier can be used to write the string in a fixed-width column. If the string is shorter than `n` characters, spaces will be added to fill the field. When `n` is positive, the spaces are added on the left; that is, the string is **right justified** in a field of `n` columns. If `n` is negative, the string is left justified. If the string is longer than `n` characters, the entire string is printed and extends beyond the specified field width. In the following example, assume that `fname` is `"Andrew"`:

```
printf( ">>%10s<<\n", fname );
printf( ">>%-10s<<\n", fname );\\
```

Then the output would be

```
>>    Andrew<<
>>Andrew    <<
```

Notes on Figure 12.10. String literals and string output. In this program, we demonstrate a collection of string-printing possibilities.

First box: ways to print strings.

- On the first line, we use `puts()` to print the contents of a string variable.
- The second line calls `printf()` to print a literal string. When printing just one string with `printf()`, that string becomes the format, which always is printed. This line also shows how escape characters are used to print quotation marks in the output.
- The third line prints both a string and a character. The `%s` in the format prints the string `"Print this string and ring the bell."` and then goes to a new line; the `%c` prints the character code `\a`, which results in a beep with no visible output on our system.
- The output from this box is

```
String demo program.
        This prints single ’ and double " quotes.
        Print this string and ring the bell.
```

- In C++ there is only one way to output a string: the operator `<<`. Figure 12.11, shows the same demo program translated to C++. Lines 13–15 correspond to Box 1 of the C program. The output from the C++ version is shown below the code.

Second box: adjacent strings are united.

- The code in Box 2 corresponds to lines 17–20 of the C++ version.
- The `puts()` shows that two strings, separated only by whitespace, become one string in the eyes of the compiler. This is a very useful technique for making programs more readable. Whitespace refers to any nonprinting character that is legal in a C source code file. This includes ordinary space, newline, vertical and horizontal tabs, formfeed, and comments. If we were to write a comma between the two strings, only the first one would be used as the format, the second one would cause a compiler error.
- The output from this box is

```
These two quoted sections form a single string.

You can break a format string into pieces if you
end each line and begin the next with quotes.
You may indent the lines any way you wish.
```

- When we have a long format in a `printf()` statement, we just break it into two parts, being sure to start and end each part with a double quote mark. Where we break the format string and how we indent it have no effect on the output.

Third box: escape characters in strings.

- The code in Box 3 corresponds to lines 22–24 of the C++ version.
- The output from this box is shown below a tab line so you can see the reason for the unusual spacing:

```
This    string  has     tab     characters      between words.
Each word on the line above starts at a tab stop.

This puts()
        will make 3 lines of output,
        indented.
```

- Note that the tab character does not insert a constant number of spaces, it moves the cursor to the next tab position. Where the tab stops occur is defined by the software (an eight-character stop was used here). In practice, it is a little tricky to get things lined up using tabs.
- The presence or absence of newline characters controls the vertical spacing. The number of lines of output does not correspond, in general, to the number of lines of code.

Fourth box: using field-width specifiers with strings.

- The output is

```
>>left justify, -field width          <<
>>         right justify, +field width<<
>>not enough room?  Field gets expanded<<
```

- Spaces are used to pad the string if it is shorter than the specified field width. Using a negative number as a field width causes the output to be left justified. A positive field width produces right justification.
- If the string is longer than the field width, the width specification is ignored and the entire string is printed.
- Formatting in C++ is very different from C because the output operator does not use formats. Instead, syntactic units called "manipulators" are written in the output stream, as if they were values to print. Lines 26–28 show how to set the field width and justification (right or left).

  Line 29 uses `endl`. This manipulator prints a newline and flushes the output from the internal buffer to the screen.

12.2.2 String Input in C

String input is more complicated than string output. The string input functions in the C standard I/O library offer a bewildering variety of options; only a few of the most useful are illustrated here. One thing common to all C string input functions is that they take one string argument, which should be the name of a character array or a pointer to a character array. *Do not use a variable of type* `char*` *for this purpose unless you have previously initialized it to point at a character array.*

For example, the diagram in Figure 12.12 shows two before-and-after scenarios that cause trouble. The code fragment on the left is wrong because the pointer is uninitialized, so it points at some random memory location; attempting to store data there is likely to cause the program to crash. The code on the right is wrong because C does not permit a program to change a literal string value. The diagrams in Figure 12.13 show two ways to do the job right. That on the left uses the name of an array in the `scanf()` statement; that on the right uses a pointer initialized to point at an array.

Also, an array that receives string input must be long enough to contain all the information that will be read from the keyboard plus a null terminator. The input, once started by the user hitting the Enter key, continues until one of two conditions occurs:

1. The input function reads its own termination character. This varies, as described next, according to the input function.
2. The maximum number of characters is read, as specified by some field-width limit.

After the read operation is over, a null character will be appended to the end of the data.

Both these `scanf()` statements are errors, The first one stores the input at some random memory location. The second one will not compile because it is attempting to change a string literal.

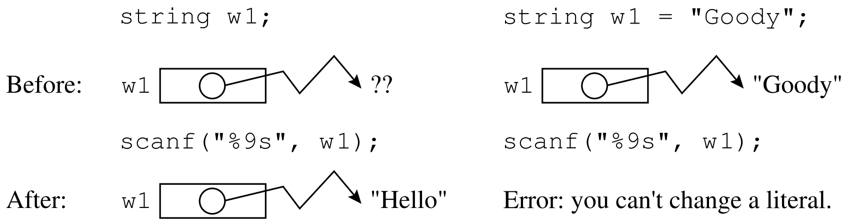


Figure 12.12. You cannot store an input string in a pointer.

Reading a string with `scanf()`. The easiest way to read a string is by using `scanf()` with a `%s` **conversion specifier**. This will skip over leading whitespace characters, then read and store input characters at the address given, until whitespace is encountered in the input. Thus, `%s` will read in only one word.

To avoid security problems, a number (the length of your array -1) must be written between the `%` and the `s` in your format. If the read operation does not stop sooner because of whitespace in the input stream, it will stop after reading `n` input characters and store a null terminator after the `n`th char. For example, `%9s` will stop the read operation after nine characters are read; the corresponding array variable must be at least 10 bytes long to allow enough space for the input plus the null terminator. Here are examples of calls on `scanf()` with and without a field-width limit. The correct call is first, the faulty one second.

```
char name[10];
scanf( "%9s", name );    // safe
scanf( "%s", name );     // dangerous; overflow possible.  DO NOT do this.
```

Note that there is no ampersand before the variable name; the `&` must be omitted because `name` is an array.

The brackets conversion. If the programmer wants some character other than whitespace to end the `scanf()` read operation, a format of the form `%`n`[^?]` may be used instead of `%`n`s` (n is still an integer denoting the field length). The desired termination character replaces the question mark and is written inside the brackets following the carat character, `^`.[6] Some examples:

```
scanf( "%29[^\n]", street );
scanf( "% 29[^,], %2[^\n]", city, state );
```

A complete input would be:

```
122 E. 42nd St.
New York, NY
```

[6] A variety of effects can be implemented using square brackets. We present only the most useful here. The interested programmer should consult a standard reference manual for a complete description.

Here are two correct and meaningful ways to read a string into memory:

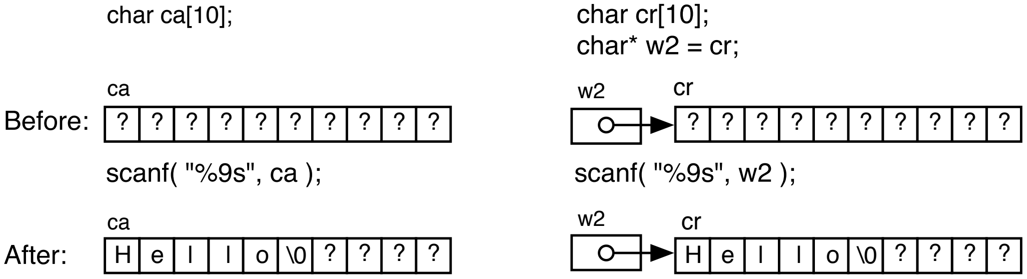


Figure 12.13. You need an array to store an input string.

The first line reads the street address and stops the read operation after 29 characters have been read, or sooner if a newline character is read. It does not stop for other kinds of whitespace. The second example reads characters into the array `city` until a comma is read. It then reads and discards the comma and reads the remaining characters into the array `state` until a newline is entered.

Formats for `scanf`. An input format contains a series of conversion specifiers. It can also contain blanks, which tell `scanf()` to ignore whitespace characters in that position of the input. Anything in an input format that is not a blank or part of a conversion specifier *must* be present in the input for correct operation. At run time, when that part of the format is interpreted, characters are read from the input buffer, compared to the characters in the format, and discarded if they match. If they do not match, it is an input error and they are left in the buffer[7].

In the second example above, the program reads two strings (city and state) that are on one input line, separated by a comma and one or more spaces. The comma is detected first by the brackets specifier but not removed from the buffer. The comma and spaces then are read and removed from the buffer because of the `", "` in the format. Finally, the second string is read and stored. These comma and whitespace characters are discarded, not stored in a variable. Whenever a non-whitespace character is used to stop an input operation, that character must be removed from the buffer by including it directly in the format or through some other method. Figure 12.14, in the next section, shows these calls on `scanf()` in the context of a complete program.

The `gets()` function C supports another string input function called `gets()` (get-string) with the prototype `string gets ( char s[] )`. It reads characters (including leading whitespace) from the input stream into the array until the first `\n` is read. The newline character terminates reading but is *not* stored in `s`. Instead, a null terminator is stored at the end of the input characters in the array. The variable `s` must refer to an array of `char`s long enough to hold the entire string and the null character.

```
char course_name[20];
gets( course_name );    // Error if input > 19 chars.
```

At first sight, it would seem that `gets()` is a good tool; it is simple and it reads a whole line without the fuss of making a format. However, this function has a fatal flaw that makes it inappropriate for use by responsible programmers: there is no way to make the read operation stop when it reaches the end of the array. If the input is too long, it will simply overwrite memory and wipe out the contents of adjacent memory variables. We call this *"walking on memory"*, and it has been the basis of countless viruses and attacks. For this reason, responsible programmers avoid `gets()` and we do not deal further with it in this text.

12.2.3 String Input in C++

The input operator in C++ is `>>`. To input something from the keyboard, we use `cin >>`. These are the most important things to know about input using `cin >>`:

- The type of value read is determined by the type of the variable you read it into. No format is needed.
- Built-in input methods exist for all the numeric types, characters, and strings.
- When using `>>` to read something, leading whitespace will be skipped by default[8].
- A string input ends when the first whitespace happens. You cannot read an entire name or sentence this way.
- When reading strings with `>>`, it is possible to read more characters than will fit into a char array. For this reason, it is *unsafe and unprofessional* to read a string with `>>`.

[7] This is another error condition that will be discussed in Chapter 14.

[8] The default can be changed but that is uncommon.

Thus, we need some other safe way to read strings in C++. The library provides three functions (with multiple versions) for this task. One is named `get())` and the other two named `getline()`. None of them skip leading whitespace, and all of them put a null terminator on the end of the string that was read in.

- `getline( cin, stringVar )`:
  This function is used to read a string of any length into a C++ string variable. The variable will "grow", as needed, to store a string of any length. The read operation stops when the next newline character is entered.
- `getline( cin, stringVar, charDelim )`:
  When a third parameter is used to call `getline()`, it is used instead of `\n` as the delimiter to end of the input. This allows you to read an entire file in one line.
- `cin.getline( charArray, maxSize )`:
  Characters are read from cin and stored in the array until the newline character is found or the maxSize-1 has been reached. A null terminator is stored in the last position. The newline is removed from the stream and discarded. It is not stored in the array.
- `cin.getline( charArray, maxSize, charDelim )`: Characters are read from cin and stored in the array until the delimiter character is found or the maxSize-1 has been reached. A null terminator is stored in the last position. The delimiter is removed from the stream and discarded. It is not stored in the array.
- `cin.get( charArray, maxSize ) and cin.get( charArray, maxSize, charDelim )`:
  This function is exactly like `getline()` except that it leaves the delimiter in the stream instead of discarding it. That char must be handled by some future read operation.

12.2.4 Guidance on Input

In general, you should use a C++ string and `getline( cin, stringVar )` for all string input. One of the plagues of mankind is malware, and many of the vulnerabilities that make malware possible are caused by inappropriate string input operations that can overflow their arrays and start overlaying adjacent memory. UsingC++ string and `getline( cin, stringVar )` avoids having this kind of vulnerability in your code. It is also a great deal easier than using arrays and pointers.

12.3 String Functions

The C and C++ libraries provide a library of functions that process strings. In addition, programmers can define their own string functions. In this section, we first examine how strings are passed as arguments and used as parameters in functions. Then we explore the many built-in string functions and how they can be used in text processing.

In the C++ code examples, you will note that functions are called in the OO manner, by putting the name of an object (and a dot) before the name of the function.

12.3.1 Strings as Parameters

The program example in Figure 12.14 demonstrates the proper syntax for a function prototype, definition, and call with a string parameter. Figure 12.15 repeats the same demonstration in C++.

Notes on Figure 12.14. A string parameter.

First box: string input.

1. In the C++ version, lines 15..24 correspond to Box 1.
2. The safe methods of string input described in the previous section are used here in a complete program. In every case, a call on `scanf()` limits the number of characters read to one less than the length of the array that receives them, leaving one space for the null terminator.

3. In the first call, the argument to `scanf()` is a string variable initialized to point to a character array. It is the correct type for input using either the `%s` or `%[^?]` conversion specifiers.
4. In the other two calls, the argument is a character array. This also is a correct type for input using these specifiers.

Second box: character and string output.

1. In the C++ version, lines 26–30 correspond to Box 2.
2. Both `putchar()` and `puts()` are used to print a newline character. The arguments are different because these functions have different types of parameters. The argument for `putchar()` is the single character, `\n`, while the string `"\n"` is sent to `puts()`, which prints the newline character and adds another newline.
3. Inner boxes: calls on `print_upper()`. The programmer-defined function `print_upper()` is called twice to print upper-case versions of two of the input strings. In the first call, the argument is a string

---

This program demonstrates a programmer-defined function with a string parameter. It also uses several of the string input and output functions in a program context.

```
#include <stdio.h>
#include <string.h>

void print_upper( char* s );

int main( void )
{
    char first[15];
    char* name = first;
    char street[30], city[30], state[3];

    printf( " Enter first name: " );
    scanf( "%14s", name );                 // Read one word only
    printf( " Enter street address: " );
    scanf( " %29[^\n]", street );          // Read to end of line
    printf(" Enter city, state: " );       // Split line at comma
    scanf( " %29[^,], %2[^\n]", city, state );

    putchar( ’\n’ );
     print_upper( name );                  // Print name in all capitals.
    printf( "\n %s %s, ", street, city ); // Print street, city as entered.
     print_upper( state );                 // Print state in all capitals.
    puts( "\n" );
}

// ---------------------------- Print letters of string in upper case.
void print_upper( char* s )
{
    for (int k = 0; s[k] != ’\0’; ++k) putchar( toupper( s[k] ) );
}
```

Figure 12.14. A string parameter in C.

```
// ---------------------------------------------------------------------
//  Figure 12.15:  A string parameter.
//  Rewritten in C++ June 21, 2016
// ---------------------------------------------------------------------
#include "tools.hpp"

void printUpper( string s );

int main( void )
{
    string name;
    string street;
    char city[30], state[3];

    cout <<"\n Enter first name: ";
    cin >>name;                         // Read one word only; space delimited.
    cout <<" Enter street address: ";
    cin >> ws;
    getline(cin, street );              // Read to end of line
    cout <<" Enter city, state: ";
    cin >> ws;
    cin.getline( city, 30, ’,’ );       // Split line at comma
    cin >> ws;
    cin.getline( state, 3 );

    cout <<"\n\n";
    printUpper( name );                        // Print name in all capitals.
    cout <<"   " <<street <<", " <<city <<", "; // Print street, city as entered.
    printUpper( state );                       // Print state in all capitals.
    cout <<endl;
    return 0;
}

// ----------------------------------------------------------------------
void printUpper( string s )
{
    for( int k = 0; s[k] != ’\0’; ++k ) s[k] = toupper(s[k]);
    cout << s << endl;
}

/* Output: --------------------------------------------------------------

 Enter first name: janet
 Enter street address: 945 Center St.
 Enter city, state: Houghton, wv


JANET
   945 Center St., Houghton, WV

---------------------------------------------------------------------- */
```

Figure 12.15. A string parameter in C++.

variable. In the second, it is a character array. Both are appropriate arguments when the parameter has type `string`. We also could call this function with a literal string.

4. Sample output from this program might be

```
Enter first name: Maggie
Enter street address: 180 River Rd.
Enter city, state: Haddon, Ct

MAGGIE
180 River Rd. Haddon, CT
```

Last box: the* `print_upper()` *function.

1. In the C++ version, lines 35–39 correspond to Box 1.
2. The parameter here is type `string`. The corresponding argument can be the name of a character array, a literal string, or a string variable. It could also be the address of a character somewhere in the middle of a null-terminated sequence of characters.
3. Since a string is a pointer to an array of characters, those characters can be accessed by writing subscripts after the name of the string parameter, as done here. The `for` loop starts at the first character of the string, converts each character to upper case, and prints it. The original string remains unchanged.
4. Like most loops that process the characters of a string, this loop ends at the null terminator.

12.3.2 The String Library

The standard C **string library** contains many functions that manipulate strings. Some of them are beyond the scope of this book, but others are so basic that they deserve mention here:

- `strlen()` finds the length of a string.
- `strcmp()` compares two strings.
- `strncmp()` compares up to n characters of two strings.
- `strcpy()` copies a whole string.
- `strncpy()` copies up to n characters of a string. If no null terminator is among these n characters, do not add one.
- `strcat()` copies a string onto the end of another string.
- `strncat()` copies up to n characters of a string onto the end of another string. If no null terminator is among these n characters, add one at the end.
- `strchr()` finds the leftmost occurrence of a given character in a string.
- `strrchr()` finds the rightmost occurrence of a given character in a string.
- `strstr()` finds the leftmost occurrence of a given substring in a string.

We discuss how these functions can be used correctly in a variety of situations and, for some, show how they might be implemented.

The length of a string. Because a string is defined as a **two-part object** (recall the diagrams in Figure 12.8), we need two methods to find the size of each part. Figures 12.16 and 12.17 illustrate the use of both operations. The operator `sizeof` returns the size of the pointer part of a string in C and the size of the string object in C++, while the function `strlen()` returns the number of characters in the array part, not including the null terminator. (Therefore, the string length of the null (empty) string is 0.) Figure 12.19 shows how `strlen()` might be implemented. Figure 12.29 shows further uses of `strlen()` in a program.

```
#include <stdio.h>
#include <cstring.h>
#define N 5

int main( void )
{
    char* w1 = "Annette";
    char* w2 = "Ann";
    char w3[20] = "Zeke";

    printf( " sizeof w1 is %2i   string is \"%s\"\t strlen is %i\n",
             sizeof w1, w1, strlen( w1 ) );
    printf( " sizeof w2 is %2i   string is \"%s\"\t\t strlen is %i\n",
             sizeof w2, w2, strlen( w2 ) );
    printf( " sizeof w3 is %2i   string is \"%s\"\t strlen is %i\n\n",
             sizeof w3, w3, strlen( w3 ) );
}
```

The output of this program is

```
 sizeof w1 is  4   string is "Annette"   strlen is 7
 sizeof w2 is  4   string is "Ann"       strlen is 3
 sizeof w3 is 20   string is "Zeke"      strlen is 4
```

Figure 12.16. The size of a C string.

```
// ---------------------------------------------------------------------
//  Figure 12.17:  The size of a C++ string.
//  Rewritten in C++ June 21, 2016
// ---------------------------------------------------------------------
#include "tools.hpp"
#define N  5

int main( void )
{
    string w4( "Annabel" );

    cout <<" sizeof w4 is " <<sizeof w4 <<" --- content is " <<w4
         <<" --- length is " <<w4.length();
    cout <<"\n\n Can you explain why?\n\n" ;
}

/* Output: --------------------------------------------------------------

 sizeof w4 is 24 --- content is Annabel --- length is 7

 Can you explain why?
*/
```

Figure 12.17. The size of a C++ string.

Notes on Figure 12.16. The size of a string.

1. The variables `w1` and `w2` are strings. When the program applies `sizeof` to a string, the result is the size of a pointer on the local system. So `sizeof w1 == sizeof w2` even though `w1` points at a longer name than `w2`.
2. In contrast, `w3` is not a string; it is a `char` array used as a container for a string. When the programmer prints `sizeof w3`, we see that `w3` occupies 20 bytes, which agrees with its declaration. This is true even though only 5 of those bytes contain letters from the string. (Remember that the null terminator takes up 1 byte, even though you do not write it or see it when you print the string.)
3. To find the number of letters in a string, we use `strlen()`, string length, not `sizeof`. The argument to `strlen()` can be a `string` variable, a string literal, or the name of a character array. The value returned is the number of characters in the array part of the string, up to but not including the null terminator.
4. In C++, sizeof works exactly as it does in C. However, to find the number of letters in a string, we use `strName.length()`.

Comparing two strings. Figure 12.18 shows what happens when we try to compare two strings or `char` arrays using `==`. Unfortunately, only the addresses get compared. So if two pointers point at the same array, as with `w5` and `w6`, they are equal; if they point at different arrays, as with `w4` and `w5`, they are not equal, even if the arrays contain the same characters.

To overcome this difficulty, the C `string` library contains the function `strcmp()`, which compares the actual characters, not the pointers. It takes two arguments that are strings (pointers) and does a letter-by-letter, alphabetic-order comparison. The return value is a negative number if the first string comes before the other alphabetically, a 0 if the strings are identical up through the null characters, and a positive number if the second string comes first. We can use `strcmp()` in the test portion of an `if` or `while` statement if we are careful. Because the intuitively opposite value of 0 is returned when the strings are the same, we must remember to compare the result of `strcmp()` to 0 in a test for equality. Figure **??** shows a proper test for equality and Figure 12.20 shows how this function could be implemented.

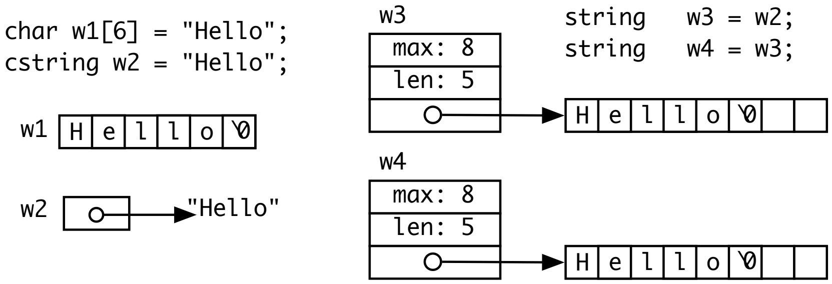
The other standard string comparison function, `strncmp()`, also deserves mention. A call takes three arguments and has the form `strncmp( s1, s2, n )`. The first two parameters are just like `strcmp()`; the third parameter is used to limit the number of letters that will be compared. For example, if we write `strncmp( "elephant", "elevator", 3 )`, the result will be 0, meaning that the strings are equal up to the $3rd$ character. If n is greater than the length of the strings, `strncmp()`, works identically to `strcmp()`.

Character search. The function `strchr( string s1, char ch )` searches `s1` for the first (leftmost) occurrence of `ch` and returns a pointer to that occurrence. If the character does not appear at all, `NULL` is returned. The function `strrchr( string s1, char ch )` is similar, but it searches `s1` for the last (rightmost) occurrence of `ch` and returns a pointer to that occurrence. If the character does not appear at all, `NULL` is returned. Figure 12.21 shows how to call both these functions and interpret their results.

Substring search. The function `strstr( string s1, string s2 )` searches `s1` for the first (leftmost) occurrence of substring `s2` and returns a pointer to the beginning of that substring. If the substring does not appear at all, `NULL` is returned. Unlike searching for a character, there is no library function that searches for the last occurrence of a substring. Figure 12.21 gives an example of using `strstr()`.

Copying a string. What does it mean to copy a string? Do we copy the pointer or the contents of the array? Both. We really need two copy operations because sometimes we want one effect, sometimes the other. Therefore, C has two kinds of operations: To copy the pointer, we use the assignment operator; to copy the contents of the array, we use `strcpy()` or `strncpy()`.

Figure 12.18 diagrams the result of using a pointer assignment; the expression `w6 = w5` copies the address from `w5` into `w6`. After the assignment, `w6` points at the same thing as `w5`. This technique is useful for output



```
// --------------------------------------------------------------------
// Figure 12.18:  Do not use == with C strings.
// Rewritten in C++ June 21, 2016
// This demo defines what == means.
// --------------------------------------------------------------------
#include "tools.hpp"
#define N  5
typedef const char* cstring;

int main( void )
{
    char w1[6] = "Hello";
    cstring w2 = "Hello";
    string w3 = w2;
    string w4 = w3;

    if (w1 == w2)   cout << "Yes, w1 == w2";
    else            cout << "No, w1 != w2";
    cout <<" because we are comparing the pointers here." <<endl;

    if (strcmp( w1, w2 ) == 0)  cout << "Yes, w1 str= w2";
    else                        cout << "No, w1 not str= w2";
    cout <<" because we are comparing the characters." <<endl;

    if (w2 == w3)   cout << "Yes, w2 == w3";
    else            cout << "No,  w2 != w3";
    cout <<" because they both point at the same thing." <<endl;

    if ( w3.compare(w4) )       cout << "w3 compares!= to w4";
    else                        cout << "w3 compares== to w4";
    cout <<" because w4 is a copy of w3.\n\n";
}
/* Output: --------------------------------------------------------------

No, w1 != w2 because we are comparing the pointers here.
Yes, w1 str= w2 because we are comparing the characters.
Yes, w2 == w3 because they both point at the same thing.
w3 compares== to w4 because w4 is a copy of w3.

*/
```

Figure 12.18. Do not use == with C strings (but it works in C++).

Walk down a string, from the beginning to the null terminator, counting characters as you go. Do not count the terminal null character.

```
int my_strlen( char * st )    // One way to calculate the string length.
{
    int k;                    // Loop counter and return value.
    for (k=0; st[k]; ++k);    // A tight loop -- no body needed.
    return k;
}
```

Figure 12.19. Computing the length of a string in C.

labeling. For example, in Figure 12.5 the value of `response` is set by assigning the address of the correct literal to it.

Copying the array portion of a string also is useful, especially when we need to extract data from an input buffer and move it into a more permanent storage area. The `strcpy()` function takes two string arguments, copies the contents of the second into the array referenced by the first, and puts a null terminator on the end of the copied string. It is necessary, first, to ensure that the receiving area is long enough to contain all the characters in the source string. This technique is illustrated in the second call of Figure 12.22. Here, the destination address is actually the middle of a character array.

The `strncpy()` function is similar to `strcpy()` but takes a third argument, an integer `n`, which indicates that copying should stop after `n` characters. Setting `n` equal to one less than the length of the receiving array guarantees that the copy operation will not overflow its bounds. Unfortunately, `strncpy()` does

Walk down two string at the same time. Leave loop if a pair of corresponding letters is unequal or if the end of one has been reached. Return a negative, zero, or positive result depending on whether the first string argument is less than, equal to, or greater than the second argument.

```
int my_strcmp( char* a, char* b )
{
    int k;
    for (k=0; a[k]; ++k){         // Leave loop on at end of string
        if (a[k] != b[k]) break;  // Leave loop on inequality
    }
    return a[k] - b[k];           // =0 if at end of both strings.
                                  // Negative if a is lexically before b.
                                  // Positive if b comes before a.
}
```

Figure 12.20. Possible implementation of `strcmp()`.

```
char line[] = "I found the cat in the fort.";
char * left, * right, * sub;
left  = strchr( line, 'n' );
right = strrchr( line, 'f' );
sub   = strstr( line, "the" );
```

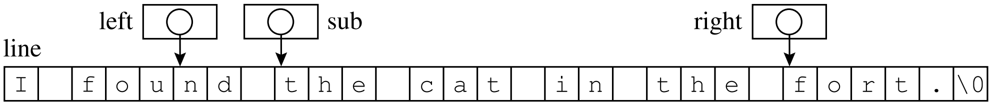
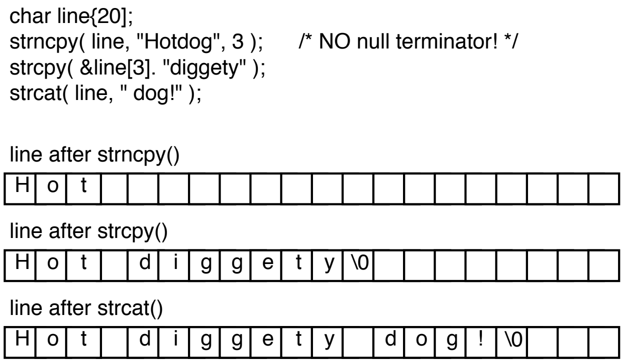


Figure 12.21. Searching for a character or a substring in C.



```
char line{20];
strncpy( line, "Hotdog", 3 );      /* NO null terminator! */
strcpy( &line[3]. "diggety" );
strcat( line, " dog!" );
```

line after strncpy()

H	o	t																	

line after strcpy()

H	o	t		d	i	g	g	e	t	y	\0								

line after strcat()

H	o	t		d	i	g	g	e	t	y		d	o	g	!	\0			

Figure 12.22. Search, copy and concatenate in C.

not automatically put a null terminator on the end of the copied string; that must be done as a separate operation unless the null character is copied by the operation. Or we can let a succeeding operation handle the termination, as happens in the example of Figure 12.22.

String concatenation. The function `strcat( string s1, string s2 )` appends `s2` onto the end of `s1`. Calling `strcat()` is equivalent to finding the end of a string, then doing a `strcpy()` operation starting at that address. If the position of the end of the string already is known, it actually is more efficient to use `strcpy()` rather than `strcat()` to append `s2`. In both cases, the array in which `s1` is stored must have enough space for the combined string. The `strcat()` function is used in the last call in the example of Figure 12.22 to complete the new string.

The function `strncat()` is like `strcat()`, except that the appending operation stops after at most `n` characters. A null terminator *is* added to the end of the string. (Therefore, up to $n + 1$ characters may be written.) The careful programmer uses `strncpy()` or `strncat()`, rather than `strcpy()` or `strcat()`, and sets `n` to the number of storage slots remaining between the end of `s1` and the end of the array.

Notes on Figure 12.23, C++ string operations. All the string function used here are in the C++ string class so they are called using the name of a string object before the function name. In the code example, the object is named "line", and it is used to call functions on lines 14–20.

- `find_first_of()`: This is like `strchr` in C, except that the return value is a subscript, not a pointer.
- `find_last_of()`: This is like `strrchr` in C, except that the return value is a subscript, not a pointer.
- `find()`: This finds a substring within a longer string. The subscript of the beginning of the leftmost occurrence is returned.
- `replace( start, len, stringVar)`: The object written to the left of the function name will be modified. The first parameter is the subscript at which the replacement should `start`. The second parameter, `llen`, tells how many characters to replace from the old string. The third parameter is the new replacement string. (If it is longer than `len`, part will not be used.)
- `append()`: The object written to the left of the function name will be modified by adding the new word on its end. The string will "grow" if needed. C
- `str::npos`: This is a constant defined in the string class, and used as an error return value. For example, if you search a string for the letter `'x'`, and it is not there, the return value will be `str::npos`. In my implementation, `str::npos` is defined as -1, however, that is not guaranteed. This value is returned from `find()` on line 16 and is displayed on line 21 of the output.

```
// ------------------------------------------------------------------
// Figure 12.23 Search, copy and concatenate in C++
// Rewritten in C++ June 21, 2016
// ------------------------------------------------------------------
#include "tools.hpp"

#define N  5

int main( void )
{
    string line = "I found chili sauce";
    int left, right, sub;

    left = line.find_first_of( ’n’ );   cout <<left <<"  ";
    right = line.find_last_of( ’f’ );   cout <<right <<endl;
    sub = line.find( "the" );           cout <<sub <<" This is string::npos. ";

    line.replace( 0, 3, "Hotdog" );
    line.replace( 3, 8, " diggety " );
    line.append( " dog!" );
    cout <<"\n" <<line <<"\n\n";
}

/* Output: -------------------------------------------------------------
5  2
-1 This is string::npos.
Hot diggety chili sauce dog!
*/
```

Figure 12.23. Search, copy and concatenate in C++.

12.4 Arrays of Strings

An important aspect of C, and all modern languages, is that aggregate types such as strings and arrays can be compounded. Thus, we can build arrays of arrays and arrays of strings. An **array of strings**, sometimes called a **ragged array** because the ends of the strings generally do not line up in a neat column, is useful for storing a list of names or messages. This saves space over a list of uniform-sized arrays that are partially empty.

We can create a ragged array by declaring an array of strings and initializing it to a list of string literals.[9] For example, the following declaration creates the data structure shown in Figure 12.24:

```
char* word[3] = \{ "one", "two", "three" \};
```

This particular form of a ragged array does not permit the contents of the entries to be changed. Making a ragged array of arrays that can be updated is explained in a later chapter. The program in Figures 12.25 and **??** demonstrate the use of ragged arrays in two different contexts.

12.4.1 The Menu Data Structure

In computer programs, a menu is a list of choices presented interactively to a computer user. Each choice corresponds to some action or process that the program can carry out and the user might wish to select. In a modern windowing system, the user may make selections using a mouse. The other standard approach is to display an alphabetic or numeric code for each option and ask the user to enter the selected code from the

[9] In a later chapter, we show how to create a ragged array dynamically.

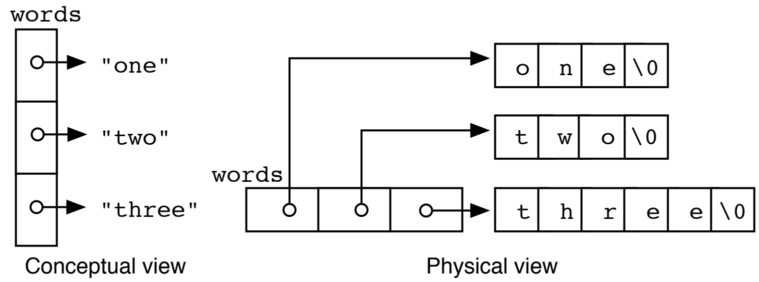


Figure 12.24. A ragged array.

keyboard. When execution of an option is complete, the menu is presented again for a new choice. Because they are easy to code and create an attractive interface, menus now are a component of programs in virtually every application area. Here, we present an easy way to set up your own menu-driven applications.

To use a menu, several things must occur:

- A title or some general instructions should be presented.
- Following that should be a list of possible choices and the input selection code for each. It is a good idea to include one option that means "take no action" or "quit."
- Then the user must be prompted to enter a choice, and the choice must be read and validated.
- Finally, the program must carry out the user's chosen intent.

Often, carrying out the chosen action means using supporting information specific to the choice. This information may be stored in a table implemented as a set of parallel arrays. In the menu display, each menu item can be shown next to its array subscript. The user is asked to choose a numbered item, and that number is used as a subscript for all of the arrays that form the table.

12.4.2 An Example: Selling Ice Cream

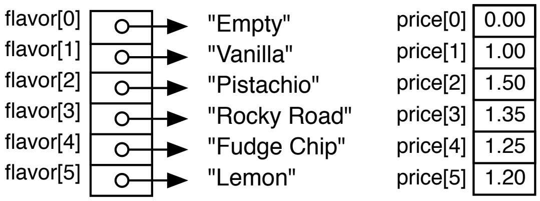
This program will be given twice, first in `C`, then in `C++` 'The `C` program in Figure 12.25 illustrates the use of a menu of ice cream flavors and a parallel array of prices, as depicted at the top. A loop prints the menu. The loop variable is used as the subscript to print each menu item and is displayed as the selector for that item. When the user enters a choice, the number is used to subscript two **parallel arrays:** the `flavor` array and the `price` array. The program then prints a message using this information (and we imagine that ice cream and money change hands).

Notes on Figure 12.25. Selecting ice cream from a menu.

First box: creating a menu.

- The first string declared here is the greeting that will appear at the top of the menu.
- Second, we declare an array of strings named `flavor` and initialize it with the menu selections we want to offer.
- Another array, `price`, parallel to `flavor`, lists the price for a cone of each flavor. The position of each price in this array must be the same as the corresponding item in `flavor`.
- Making the menu longer or shorter is fairly easy: Just change the value of `CHOICES` at the top of the program, add or delete a string to or from the `flavor` array, and add or delete a corresponding cost to or from the `price` array.
- Since this is a relatively small program, the declarations have been placed in `main()`. Alternative placements are discussed as they occur in other examples in this chapter.

This program builds the parallel-array data structure diagrammed below, then calls the `menu()` function in Figure 12.26.



```
#include <stdio.h>
#include <string.h>
#define CHOICES 6
int menu ( string title, int max, string menu_list[] );

int main( void )
{
    int choice;                                  // The menu selection.
    string greeting = " I'm happy to serve you.  Our specials today are: ";
    string flavor[CHOICES] = { "Empty", "Vanilla", "Pistachio",
                               "Rocky Road", "Fudge Chip", "Lemon" };
    float price[CHOICES] = { 0.00, 1.00, 1.50, 1.35, 1.25, 1.20 };

    choice = menu( greeting, CHOICES, flavor );

    printf( "\n Here is your %s cone.\n That will be $%.2f.  ",
            flavor[choice], price[choice] );
    puts( "Thank you, come again." );
}
```

Figure 12.25. Selecting ice cream from a menu.

Second box: calling the `menu()` ***function.***

- Since the `menu()` function, defined in Figure 12.26, will be used for all sorts of menus in various programs, the program must supply, as an argument, an appropriate application title to be displayed above the menu items.
- The second argument is the number of items in the menu. In C, the number of items in an array must be an argument to any function that processes the array; the function cannot determine this quantity from just the array itself.
- The third argument is the name of the array that contains the menu descriptions. These descriptions will be displayed for the user next to the selection codes.
- The return value from this function is the index of the menu item that the user selected. The `menu()` function returns only valid selections, so we can safely store the return value and use it as a subscript with no further checking.

Third box: the output.

- The variable `choice` is used to subscript both arrays. The program prints the chosen flavor and the price for that flavor using the validated menu selection as a subscript for the `flavor` and `price` arrays, respectively.
- Below the menu presented on the screen (shown later), the user will see his or her selection, as well as the results, displayed like this:

```
Please enter your selection: 3

Here is your Rocky Road cone.
That will be $1.35.  Thank you, come again.
```

This function is called from the main program in Figure 12.25.

```
int menu( char* title, int max, char* menu_list[] )
{
    int choice;
    printf( "\n %s\n ", title );
    for (int n = 0; n < max; ++n)  printf( "\t %i. %s \n", n, menu_list[n] );

    printf( " Please enter your selection: ");
    for(;;) {              // Prompt for and validate a menu selection.
        scanf( "%i", &choice );
        if (choice >= 0 && choice < max) break;  // Accept valid choice.
        printf( " Please enter number between 0 and %i: ", max - 1 );
    }

    return choice;
}
```

Figure 12.26. A menu-handling function.

Notes on Figures 12.26, 12.27, and 12.28. A menu-handling function.

First box: the function header.

- The first parameter is a string that contains a title for the menu. This should be declared with the menu array, as in Figure 12.25.
- The second parameter is the number of choices in the menu. This number is used to control the loop that displays the menu and determine which index inputs are legal and which are out of bounds.
- The final parameter is an array of strings that contains the list of menu items.
- The return value will be a legal subscript for the arrays that contain the menu and price information.

Second box: displaying the menu.

- First, the program prints the title for the menu with appropriate vertical spacing.
- Then it uses a loop to display the menu. On each repetition, the loop displays the loop counter and one string. This creates a numbered list of items, where each number displayed is the array subscript of the corresponding item.
- The actual menu display follows. Note that choice 0 permits an escape from this menu without buying anything. It is a good idea to include such a “no operation” alternative. A “quit” option is not necessary in this menu since it is displayed only once by the program.

```
I’m happy to serve you.  Our specials today are:
        0. Empty
        1. Vanilla
        2. Pistachio
        3. Rocky Road
        4. Fudge Chip
        5. Lemon
```

Third box: the prompt, input, and menu selection validation.

- The original prompt is written outside the loop, since it will be displayed only once. If the first selection is invalid, the user will see an error prompt.

```
// --------------------------------------------------------------------
//  Figure 12.27: Selecting ice cream from a menu in C++.
//  Rewritten in C++ June 18, 2016
// --------------------------------------------------------------------
#include "tools.hpp"

#define CHOICES  6
int menu ( const char* title, int max, const char* menuList[] );

int main( void )
{
    int choice;                                      // The menu selection
    const char* greeting = "\n I'm happy to serve you.  Our specials today are: ";
    const char* flavor[CHOICES] = { "Empty", "Vanilla", "Pistachio",
                                    "Rocky Road", "Fudge Chip", "Lemon" };
    float price[CHOICES] = { 0.00, 1.00, 1.50, 1.35, 1.25, 1.20 };

    choice = menu( greeting, CHOICES, flavor );
    cout << "\n Here is your " << flavor[choice] <<" cone.  That will be $"
        <<fixed <<setprecision(2) << price[choice] <<"\n Thank you, come again. \n";
}

// ---------------------------------------------------------------------
// Display a menu then read and validate a numeric menu choice.
int
menu ( const char* title, int max, const char* menuList[] )
{
    int choice;                 // To store the menu selection.
    printf( "\n %s\n", title );
    for (int n=0; n < max; ++n)   cout << "\t "<<n << ".  " <<menuList[n] <<endl;

    cout << " Please enter your selection: ";
    for(;;) {                  // Prompt for and validate a menu selection.
        cin >> choice;
        if (choice >= 0 && choice < max) break;     // Accept valid choice.
        cout << " Please enter a number between 0 and " << max-1 <<endl;
    }
    return choice;
}
/* Output: ------------------------------------------------------------
 I'm happy to serve you.  Our specials today are:
     0.  Empty
     1.  Vanilla
     2.  Pistachio
     3.  Rocky Road
     4.  Fudge Chip
     5.  Lemon
 Please enter your selection: 2

 Here is your Pistachio cone.  That will be $1.50
 Thank you, come again.
-------------------------------------------------
 I'm happy to serve you.  Our specials today are:
     0.  Empty
     1.  Vanilla
     2.  Pistachio
     3.  Rocky Road
     4.  Fudge Chip
     5.  Lemon
 Please enter your selection: 6
 Please enter a number between 0 and 5
---------------------------------------------------------------------- */
```

Figure 12.27. Buying a Cone in C++.

- This for loop is a typical data validation loop. It is very important to validate an input value before using it as a subscript. If that value is outside the range of legal subscripts, using it could cause the program to crash. The smallest legal subscript always is 0. The second argument to this function is the number of items in the menu array, which is one greater than the maximum legal subscript. The program compares the user's input to these bounds: If the selection is too big or too small, it displays an error prompt and asks for a new selection.
- A good user interface informs the user of the valid limits for a choice after he or she makes a mistake. Otherwise, the user might not understand what is wrong and have to figure it out by trial and error.
- When control leaves the loop, choice contains a number between 0 and max-1, which is a legal subscript for a menu with max slots. The function returns this choice. The calling program prints the chosen flavor and the price for that flavor using this number as a subscript for the flavor and price arrays.

```
// ---------------------------------------------------------------------
//  Figure 12.28: Read a menu choice and use it without validation.
//  Rewritten in C++ June 18, 2016
// ---------------------------------------------------------------------
int
menu ( const char* title, int max, const char* menuList[] )
{
    int choice;                 // To store the menu selection.
    printf( "\n %s\n", title );
    for (int n=0; n < max; ++n)   cout << "\t "<<n << ".  " <<menuList[n] <<endl;

    cout << " Please enter your selection: ";
    cin >> choice;
    return choice;
}

/* Output: --------------------------------------------------------------
 I'm happy to serve you.  Our specials today are:
     0.  Empty
     1.  Vanilla
     2.  Pistachio
     3.  Rocky Road
     4.  Fudge Chip
     5.  Lemon
 Please enter your selection: -1

 Here is your ?% ?? cone.  That will be $0.00
 Thank you, come again.
 ----------------------------------------------

 I'm happy to serve you.  Our specials today are:
     0.  Empty
     1.  Vanilla
     2.  Pistachio
     3.  Rocky Road
     4.  Fudge Chip
     5.  Lemon
 Please enter your selection: 6

Segmentation fault
---------------------------------------------------------------------- */
```

Figure 12.28. Using an invalid subscript.

- The following output demonstrates the validation process. The `menu()` function does not return to `main()` until the user makes a valid selection. After returning, `main()` uses the selection to print the chosen flavor and its price:

```
I'm happy to serve you.  Our specials today are:
        0. Empty
        1. Vanilla
        2. Pistachio
        3. Rocky Road
        4. Fudge Chip
        5. Lemon
Please enter your selection: -3
Please enter number between 0 and 5: 6
Please enter number between 0 and 5: 5

Here is your Lemon cone.
That will be $1.20.  Thank you, come again.
```

- The `menu()` function in Figure 12.26 must validate the input because C and C++ do not guard against the use of meaningless subscripts. To demonstrate the effects of using an invalid subscript, we removed the data validation loop from the third box, leaving only the prompt and the input. See Figure refex-badsub-func.

  As you can see, C++ calculates a machine address based on any subscript given it and uses that address, even though it is not within the array bounds. The results normally are garbage and should be different for each illegal input. Sometimes the output is simply garbage, as in the first example. On some computer systems, a segmentation error will probably cause the program to crash and may force the user to reboot the computer. A well-engineered program does not let an input error cause a system crash or garbage output. It validates the user's response to ensure that it is a legal choice, as in Figure 12.26.

12.5 String Processing Applications

In this section, we examine two applications that use many of the string processing functions presented in the chapter.

12.5.1 Password Validation

The following program uses some of the string functions in a practical context; namely, requiring the user to enter a password to access a sensitive database on the computer. This code could be part of many applications and will be used in Chapter 14 in a complete program.

Figure 12.29 is a main program that performs password validation, then calls an application function. In this example, the password is a constant in the main program, and it is not encrypted. In a real application, it would be encrypted and it would not be constant.

Notes on Figures 12.29 and 12.30. Checking a password.

First box : A limit on string length.

- `BUFLEN` is used to define the variable `word`, which will store the user's password input, and it is used in the call on `scanf()` to read the password from the user. This limits passwords to 79 characters plus a null terminator, an arbitrary length that is much longer than we expect to need. 80 characters is a common limit for the length of an input line. This usage can be traced back to the time when punched cards were used for computer input. A punched card had 80 columns.
- In C++, no length limit is needed because a `string` will be allocated that is big enough to hold anything that is entered.

Create a personalized form letter to send to each members of the class.

```
#include <stdio.h>
#include <string.h> // for strlen(), strcmp(), strcpy(), and strchr()
#include <ctype.h>  // for isspace()

#define BUFLEN 80
void doTask( void ){ puts( "Entering doTask function\n" ); }

// ---------------------------------------------------------------------------
int main( void )
{
    char password[] = "StaySafe";
    char word[ BUFLEN ];

    puts( "\nABC Corporation Administrative Database" );
    printf( "\nPlease enter the password: " );
    scanf( "%79[^\n]", word );

    if (strcmp( word, password ))   printf( "No Entry! \n" );
    else doTask();    // Application logic would be in this function.

    return 0;
}
```

Figure 12.29. Password validation: Comparing strings.

Second box and* C++ *lines 11–12: Defining the password.

- In a real application, the owner of the application would be able to change the password. However, we are trying to keep this program simple and so we are using a literal constant as the password. The program would need to be recompiled to change it.
- In C, character arrays are used for both the password and the input variable. The password array is 9 characters long: enough to store the quoted string and a null.
- In C++, type string is used for both the password and the input variable. We could use character arrays, just as we do in C, but strings are preferred because they are less prone to error. We do not need to worry about limiting the length of a string.

Third box and* C++ *lines 14–16: Entering the password.

- We prompt the user to enter a password. In a real application, this would be done without showing the password on the screen. In this program, however, we let the password stay on the screen for simplicity, and because we would need to go outside standard C to mask the password input.
- This call on `scanf()` will read all the characters typed, up to but not including the end-of-line character, and store them in `word`. A maximum of 79 characters will be read, to avoid overflowing the buffer. If more characters are typed, the extra ones will remain, unread, in the input stream.
- In C++, we want to read into a string, not a char array so we do not use `cin >>`. The form of `getline` with two parameters reads from an input stream into a string.

Fourth box and* C++ *lines 18–19: the comparison.

- `strcmp()` takes two C string arguments of any variety. Here, the arguments are both character arrays, but one could be a literal string.
- If the two strings are unequal, `strcmp()` will return a non-zero value and and an error comment will be printed.

- If the two strings are equal, the `else` clause will be selected. The user now has gained entry into the protected part of the program, which calls the `doTask()` function.
- In C++, it is possible to use `==` to compare the content of two C++ strings. This is much less error-prone than the old C version. This is a good reason to use C++ strings in your programs instead of C strings.
- The C++ program shows output from successful and unsuccessful login attempts.

12.5.2 The `menu_c()` Function

The process of displaying the menu and eliciting a selection is much the same for any menu-based program: the content of the menu changes from application to application but the processing method remains the same. There is one major variation: some menus are character-based and the menu choice is type `char`, (not type `int`) and is processed by a `switch` (not by using subscript). The commonality from application to application enables us to write two general-purpose menu-handling function:`menu()`, for integer choices, is shown above in Figure 12.26, and `menu_c()`, for character choices, is in Figure 12.31.

Like the `menu()` function, `menu_c()` receives a menu title, the number of menu items, and the menu array as arguments. Now, however, each menu string now consists of a selection code character and a phrase

```
// ---------------------------------------------------------------------
//  Figure 12.29: Checking a password in C++.
//  Rewritten in C++ June 24, 2016
// ---------------------------------------------------------------------
#include "tools.hpp"
void doTask( void ){ cout <<"Entering doTask function" <<endl; }

// ------------------------------------------------------------------
int main( void )
{
    string password = "StaySafe";
    string word;

    cout<<( "\nABC Corporation Administrative Database"
            "\nPlease enter the password: " );
    getline( cin, word );

    if ( word == password ) doTask();
    else cout <<"No Entry!" <<endl;

    return 0;
}

/* Output: ------------------------------------------------------------

ABC Corporation Administrative Database
Please enter the password: StaySave
No Entry!
----------------------------------------

ABC Corporation Administrative Database
Please enter the password: StaySafe
Entering doTask function
*/
```

Figure 12.30. Password validation: Comparing strings.

This function displays a menu, then reads, validates, and returns a non-whitespace character. It is called from Figures 12.34.

```
char menu_c( string title, int n, const string menu[], string valid )
{
    char ch;
    printf("\n%s\n\n", title);
    for(;;) {
        for( int k=0; k<n; ++k ) printf("\t%s \n", menu[k]);
        printf("\n Enter code of desired item: ");
        scanf(" %c", &ch);
        if (strchr( valid, ch )) break;

        while (getchar() != ’\n’);     // Discard rest of input line
        puts("Illegal choice or input error; try again.");
    }
    return ch;
}
```

Figure 12.31. The `menu_c()` function.

describing the menu item. The fourth argument is a string consisting of all the letters that are legal menu choices, and is used to validate the selection.

The function displays the strings from the menu array, one per line, then reads and returns the menu choice in the form of a single nonwhitespace character.[10] This function will be used in the next program, Figure 12.34.

Notes on Figures 12.31 and 12.32. The `menu_c()` and `doMenu()` functions. We expect the third parameter (the menu) to have a phrase describing each choice and, with that phrase, a letter to type.

[10] In contrast, `menu()` reads and returns an integer response that can be used directly as a table subscript.

```
// ---------------------------------------------------------------------------
//  Figure 12.32:  Read and validate an alphabetic menu choice character.
// ---------------------------------------------------------------------------
char
doMenu( cstring title, int n, cstring menItems[], string valid )
{
    char ch;
    cout <<"\n" <<title <<"\n\n";
    for(;;) {
        for( int k=0; k<n; ++k ) cout <<"\t" <<menItems[k] <<"\n";
        cout <<"\n Enter code of desired item: ";
        cin >>ch;
        if ( valid.find_first_of(ch) !=  string::npos ) break;
        cin.ignore(255);            // Skip inputs up to first newline.
        cout <<"Illegal choice or input error; try again.\n";
    }
    return ch;
}
```

Figure 12.32. `menu_c` **in** C++.

This function displays a title and a menu and prompts for a response in the same manner as `menu()`, in Figure 12.26. After validation, the response is returned.

First box: Reading and validating the choice.

- After the for loop displays the menu, the `printf()` prompts the user to make a selection. It lets the user know that a character (not a number) is expected this time. This is almost the same as lines l11 and 12 in the C++ version
- We use `scanf()` with `" %c"` to read the user's response. The space before the `%c` is important because the input stream probably contains a newline character from some prior input step.[11] The program must skip over that character to reach the user's current input.

  Line 13 of the C++ version dos this task. It is much simpler because `>>` always skips leading whitespace.
- To validate the user's choice, we compare it to the list of valid choices that were supplied as the fourth parameter. The `strchr()` function makes this very easy: if the user's choice is in the list, `strchr()` will return a pointer to that letter, If it is `not` in the list, `strchr()` will return a `NULL` pointer. Thus, a non-null return value indicates a valid choice. We use the `if` statement to test this value. If the choice is valid, control leaves the loop. Otherwise, we move on to the error handling code.

  In C++, the `find_first_of()` function makes the same search, but it returns a subscript, not a pointer. If the input char is not in the list of legal inputs, the function returns `str::npos`. Any other return value indicates a valid input choice.

Second box: Error handling.

- This is a one-line loop with no loop body; the semicolon is not a mistake. It reads a character and tests it, reads and tests, until the newline is found. The purpose is to skip over any extra characters that the user might have typed and that might be still in the input stream. Cleaning out the input stream is often done after an error to help the user become synchronized with the action of the program and what should happen next.

  Line 15 of the C++ version does this job using a built-in stream manipulator, `ignore`. This line will remove up to 255 chars from the input stream, but stops when a newline is found.
- We announce clearly that an error was made. Control will return to the top of the loop and display the menu again. The only way to leave the loop is to enter a valid choice. This is done on line 16 of the C++ version.

12.5.3 Menu Processing and String Parsing

The `switch` statement was introduced back in Section 6.3 as a method of decision making. Putting a `switch` inside a loop is an important control pattern used in menu processing, especially when the menu choices are characters rather than numbers. The loop is used to display the menu repeatedly, until the user wishes to quit, while the `switch` is used to process each selection. One option on the menu should be to quit, that is, to leave the loop.

This example program starts with a flow diagram of the main function, followed by the main function in C, the main function in C++, and notes on these three Figures. Following that are the subfunctions in both languages and notes on the subfunctions.

Notes on Figures 12.33, 12.34 and 12.35. Magic memo maker and its flow diagram. This flow chart is the same for both C and C++ versions.

[11] Review the discussion of these issues concerning problems with `getchar()` following Figure 8.6.

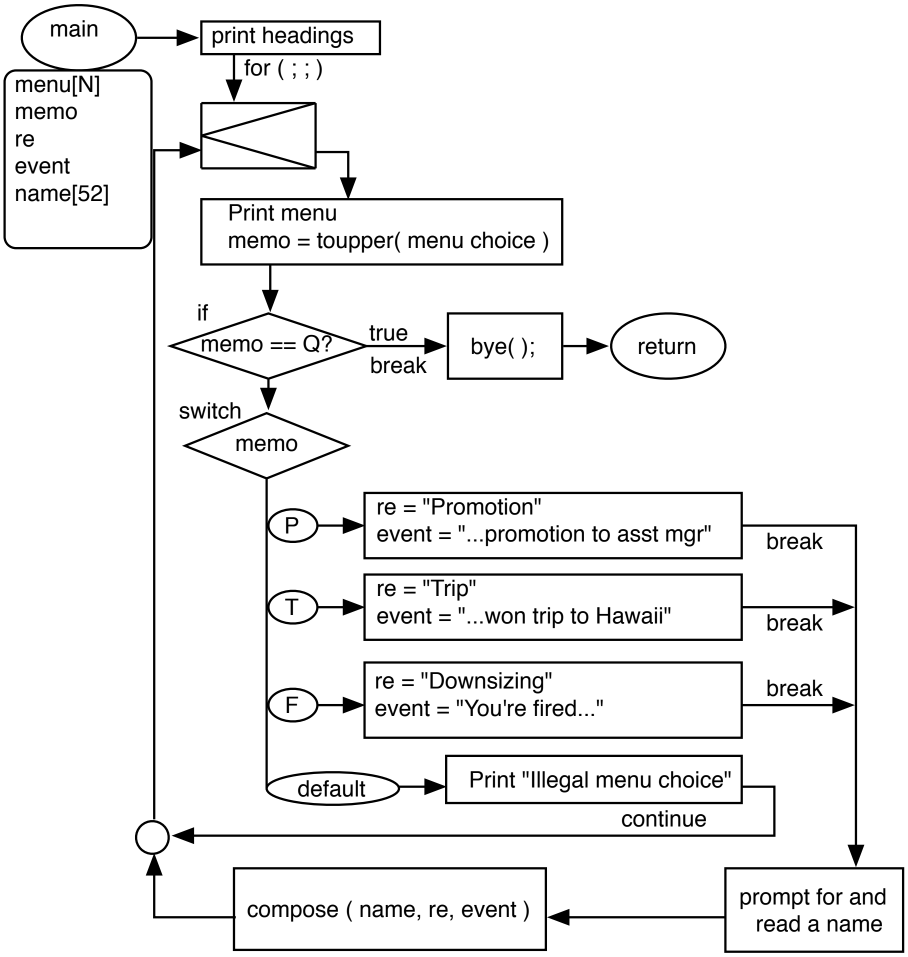
First box: the menu.

- This box corresponds to lines 8–12 in the C++ version.
- As usual, a menu consists of an integer, `N`, and a list of `N` strings to be displayed for the user. This menu is designed to be used with `menu_c()`, so each string incorporates both a single-letter code and a brief description of the option.
- To add another menu option, we must increase `N`, add a string to the array, and add another case in the switch statement (second box). To make such modifications easier, the declarations have been placed at the top of the program. The `const` qualifier is used to protect them from accidental destruction, since they are global.

Second box: the menu loop.

- This box corresponds to lines 23–45 in the C++ version.
- This menu-processing loop is typical and can be adapted to a wide variety of applications. Its basic actions are

  An example of this control structure is given in Figures 12.34 and 12.33. The program in Figure 12.34 presents a menu and processes selections until the user types the code `q` or `Q`. It illustrates the use of a



main
print headings
for (; ;)
menu[N]
memo
re
event
name[52]
Print menu
memo = toupper(menu choice)
if
memo == Q?
true
break
bye( );
return
switch
memo
P
re = "Promotion"
event = "...promotion to asst mgr"
break
T
re = "Trip"
event = "...won trip to Hawaii"
break
F
re = "Downsizing"
event = "You're fired..."
break
default
Print "Illegal menu choice"
continue
compose (name, re, event)
prompt for and read a name

Figure 12.33. Flow diagram for Figure 12.34.

This program calls the `compose()` function from Figure 12.37. Its control flow is diagrammed in Figure 12.33.

```
#include <stdio.h>
#include <string.h> // For strrchr().
#include <ctype.h>  // For toupper().

void compose( char* name, char* re, char* event );  // Modifies name.

#define N 4         // Number of memo menu choices
const char* menu[N] = {"P  Promote", "T  Trip", "F  Fire", "Q  Done"};

int main( void )
{
    char memo;         // Menu choice.
    char *re, *event;  // Subject and main text of memo.
    char name[52];     // Employee’s complete name.

    puts( "\n Magic Memo Maker" );

    for(;;) {
        memo = toupper( menu_c( " Next memo:", N, menu ) );
        if (memo == ’Q’) break;    // Leave for loop and end program.

        switch (memo) {
          case ’P’: re = "Promotion";
                    event = "You are being promoted to assistant manager.";
                    break;
          case ’T’: re = "Trip";
                    event = "You are tops this year "
                            "and have won a trip to Hawaii.";
                    break;
          case ’F’: re = "Downsizing";
                    event = "You’re fired.\n "
                            "Pick up your final paycheck from personnel.";
                    break;
        }

        printf( " Enter name: " );
        scanf( " %51[^\n]", name );
        compose( name, re, event );
    }
    return 0;
}
```

Figure 12.34. Magic memo maker in C.

```
// ---------------------------------------------------------------------------
//  Figure 12.32:  Magic memo maker in C++.
//  Rewritten in C++ June 21, 2016
// ---------------------------------------------------------------------------
#include "tools.hpp"
typedef const char*  cstring;

#define N  4                // Number of memo menu choices
#define BOSS "Leland Power"
cstring menItems[] = {"P  Promote", "T  Trip", "F  Fire", "Q  Done"};
const string valid( "PpTtFfQq" ); // The valid menu choices for this app.

char doMenu( cstring title, int n, cstring  menItems[], string valid );
void compose( string  name, cstring re, cstring event );

int main( void )
{
    char memo;                  // Menu choice.
    cstring re, event;     // Subject and main text of memo.
    string name;                // Employee’s complete name.

    cout <<"\n Magic Memo Maker\n";
    for(;;) {
        char ch = doMenu( " Next memo:", N, menItems, valid );
        memo = toupper(  ch );
        if (memo == ’Q’) break;   // Leave for loop and end program.

        switch (memo) {
          case ’P’: re = "Promotion";
                    event = "You are being promoted to assistant manager.";
                    break;
          case ’T’: re = "Trip";
                    event = "You are tops this year "
                            "and have won a trip to Hawaii.";
                    break;
          case ’F’: re = "Downsizing";
                    event = "You’re fired. \n "
                            "Pick up your final paycheck from personnel.";
                    break;
        }
        cout <<" Enter name: ";
        cin >> ws;
        getline( cin, name );
        compose( name, re, event );
    }
    return 0;
}
```

Figure 12.35. Magic memo maker in C++.

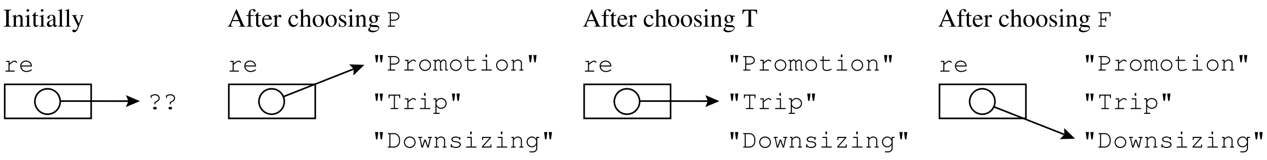


Figure 12.36. Using a string variable with a menu.

`for` loop with a `switch` statement inside it to process the selections, a `continue` statement to handle errors, and a `break` statement to quit. This program also shows a use of `strchr()` and `strrchr()` in context.

- The first inner box inn C and lines 24–26 in C++ read the user's choice. We use `menu_c()` or `coMenu()` to display a menu and return a validated selection. Then we convert the selection to upper case and break out of the `for` loop if the user has chosen the "Quit" option.
- The second inner box (lones 28–40 in C++ receives the choice and we use a `switch` to process the three legal action codes. A case is defined for each one that sets two variables that will be used later to print parts of the message. Figure 12.36 shows how we set the value of the string variable `re` for the memo generator:

  The `break` at the end of each valid case takes control to the processing statements at the bottom of the loop.
- We do not need a default case in this program because the `menu_c()` function validates the input and supplies it for the switch.
- The request is processed in the third inner box (lines 41–44 in C++). The common actions for generating the memo are performed after the end of the `switch`. When control reaches this point, invalid menu choices have been eliminated and valid ones fully set up. In this program, we call the `compose()` function to process the request.

The output from the main program begins like this:

```
Magic Memo Maker
Next memo:

        P  Promote
        T  Trip
        F  Fire
        Q  Done

Enter letter of desired item: T
Enter name: Harvey P. Rabbit, Jr.
```

In Figure 12.37, the memo is composed and processed.

Notes on Figure 12.37. Composing and printing the memo.

First box: memo header.

- This box corresponds to C++ lines 57–58.
- The program prints a memo header that contains the employee's entire name, the boss's name, and the subject of the memo.
- The boss's name is supplied by a `#define` command. The other information is passed as arguments from `main()`.

Second box: finding the end of the last name.

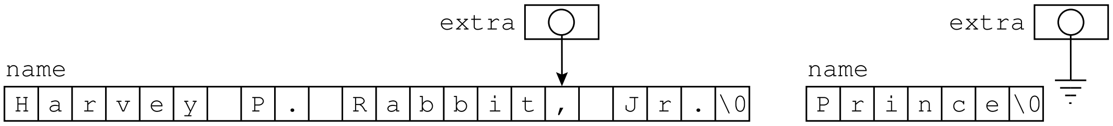
- This box corresponds to the C++ version, lines 60 and 61. The string parameter is modified in the C version but in C++, parts of it are copied to other variables instead.
- A name can have up to four parts: first name, middle initial (optional), last name, and a comma followed by a title (optional).
- For the salutation, we wish to print only the employee's last name, omitting the first name, middle initial, and any titles such as Jr. or V.P. Since the initial and titles are optional, it is a nontrivial task to *locate* the last name and separate it from the rest of the name. The last name is followed by either a comma or the null terminator that ends the string.
- In C++, we search the name for a comma thus:

  ```
  extra = name.find_first_of( ',' );              // Find end of the last name.
  if (extra != string::npos) lname = name.substr(0,extra);
  ```

  If the result is `str::npos`, we know that the last name is the last thing in the string. We then make a copy of the head of the string, up to the comma at the end of the last name. This isolates the name and eliminates the extra part.
- In C, we search the name for a comma thus:

  ```
  extra = strchr( name, ',' );
  if (extra != NULL) *extra = '\0';
  ```

  If the result is `NULL`, we know that the last name is the last thing in the string. Otherwise, the result of `strchr()` is a pointer to the comma. The following diagram shows how the `extra` pointer would be positioned for two sample inputs, with and without "extra" material at the end:



Store \0 in place of comma.

We replace the comma, if it exists, with a null character, using `*extra = '\0'`. We now know for certain that the last name is followed by a null character. This has the side effect of changing the value of the first argument of `compose()` by shortening the string. The part cut off is colored gray in the next diagram. This is an efficient but generally risky technique. Documentation must be provided to warn programmers not to call the function with constants, literal strings, or strings that will be used again. For a safe alternative, use the C++ version, where the nMW is copied into a local variable and the original remains unchanged.

Third box: the beginning of the last name.

- This corresponds to lines 63 and 64 of the C++ version.
- To find the beginning of the last name in the (newly shortened) string using C++, call `lname.find_last_of( ' ' )` The result is the subscript of the rightmost space in the lname. If a space is found, the last name will be between that space and the end of the string. The program then uses the `substr()` function and assignment to copy the last name into lname. You can safely assign one C++ string to another.

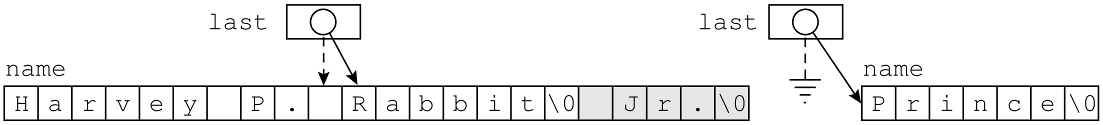
  If the name has only one part, like Midori or Prince, the result stored in `last`, will be `str::npos`. In that case, the last name is the first and only thing in the string and is ready to use.
- To find the beginning of the last name in C, use `strrchr(name, ' ')` to search the (newly shortened) string for the rightmost space. The result is indicated by a dashed arrow in the diagram that follows.

This function is called from Figure 12.34. As a side effect, the first argument, name, is modified.

```
#define BOSS  "Leland Power"

void compose( char* name, char* re, char* event )
{
    char* extra, * last;            // Pointers used for parsing name.
    printf( "\n\n To: %s\n", name );
    printf( " From: The Big Boss\n" );
    printf( " Re: %s\n\n", re );

    extra = strchr( name, ',' );       // Find end of last name.
    if (extra != NULL) *extra = '\0';  // Mark end of last name.

    last = strrchr( name, ' ' );       // Find beginning of last name.
    if (last == NULL) last = name;
    else ++last;

    printf( " Dear M. %s:\n %s\n\n -- %s\n\n", last, event, BOSS );
}
```

Figure 12.37. Composing and printing the memo in C.

```
last = strrchr( name, ' ' );
if (last == NULL) last = name;
else ++ last;
```



- If a space is found, as in the diagram on the left, the last name will be between that space and the (newly written) null terminator. The program then increments the pointer one slot so that it points at the first letter of the name.
- If the name has only one part, like Midori or Prince, the result of strrchr(), which is stored in last, will be NULL. In that case, the last name is the first and only thing in the string, so we set last = name.

Fourth box and line 66: printing the memo. The program now has located the last name and is ready to print the rest of the memo. The event (an input parameter) and the boss's name (given by a #define command) are used. Here is a sample output memo:

```
To: Harvey P. Rabbit, Jr.
From: The Big Boss
Re: Trip

Dear M. Rabbit:
You are tops this year and have won a trip to Hawaii.

-- Leland Power
```

```
// --------------------------------------------------------------------------------
//  Figure 12.35:  Composing and printing the memo in C++.
// --------------------------------------------------------------------------------
void
compose( string name, cstring re, cstring event )
{
    int extra, last;                        // Subscripts used for parsing name.
    string lname = name;

    cout <<"\n\n To: " <<name <<"\n From: The Big Boss\n";
    cout <<" Re: " <<re <<"\n\n";

    extra = name.find_first_of( ’,’ );             // Find end of the last name.
    if (extra != string::npos) lname = name.substr(0,extra); // eliminate extra.

    last = lname.find_last_of( ’ ’ );        // Find beginning of the last name.
    if (last != string::npos) lname = lname.substr(last+1);

    cout << " Dear M. " <<lname <<":\n " <<event <<"\n\n -- " <<BOSS <<"\n\n";
}



/* Output: -----------------------------------------------------------------------

 Magic Memo Maker

 Next memo:

    P  Promote
    T  Trip
    F  Fire
    Q  Done

 Enter code of desired item: p
 Enter name: Anden


 To: Anden
 From: The Big Boss
 Re: Promotion

 Dear M. Anden:
 You are being promoted to assistant manager.

 -- Leland Power
-------------------------------------------------------------------------------- */
```

Figure 12.38. Composing and printing the memo in C++.

12.6 What You Should Remember

12.6.1 Major Concepts

- The type name string is not a built-in name in C; use a char array when you want to store a string, and use a char* to point at a string in an array.
- The type name string IS standard in C++. The old C types also work in C++ and must be used for a variety of purposes.
- It is not possible to change the contents of a string literal.
- C's standard string library contains many useful functions for operating on strings. In this chapter, we note the following:
  - To find the number of characters in a string, use strlen(). (Note that sizeof gives the size of the pointer part.)
  - To search a string for a given character or substring, use strchr(), strrchr(), and strstr().
  - To compare two strings for alphabetic order, use strcmp() and strncmp(). (Note that == tells only whether two string pointers point at the same slot in an array.)
  - To copy a string or a part of a string, use strcpy(), strncpy(), strcat(), and strncat(). (Note that = copies only the pointer part of the string and cannot be used to copy an array.)
- C++'s standard string library a similar collection of useful functions for operating on C++ strings. In this chapter, we have used the following:
  - To find the number of characters in a string, use t͡length().
  - To search a string for a given character or substring, use find_first_of(), find_last_of(), and find().
  - To compare two strings for alphabetic order in C++, use compare().
  - To copy a string into a string variable, use =. To copy a part of a string, use replace() or append().
- A C string or a character array may be initialized with a string literal. The length of the array must be at least one longer than the number of letters in the string to allow for the null terminator.
- A C++ string may be initialized with any string literal of any length.
- Literal strings, char* variables, and character arrays all are called *cstrings*. However, these objects have very different properties and are not fully interchangeable.
- In memory, a cstring is represented as a pointer to a null-terminated array of characters. You can point at any character in the array.
- A C++ string is a compound data object with two integer parts and a cstring part.
- A string variable is a pointer and must be set to point at some referent before it can be used. The referent can be either a quoted string or a character array.
- The operators that work on the pointer part of a C string are different from the functions that do analogous jobs on the array part. Both are different from the operators that work with C++ strings.
- An array of strings, or ragged array, is a compound data structure that often saves memory space over an array of fixed-length strings. One common use is to represent a menu.
- An essential part of interactive programs is the display of a menu to provide choices to the user. It is essential to validate menu choices before using them.

  A switch statement embedded in a loop is a common control structure for processing menus. The menu display and choice selection continue until the user picks a termination option.

12.6.2 Programming Style

- A loop that processes all of the characters in a string typically is a sentinel loop that stops at the null terminator. Many string functions operate in this manner.
- Use the "zero" literal of the correct type. Use `NULL` for pointers in C, `\0` for characters, and `""` for strings. Reserve `0` for use as an integer. Use `nullptr` for pointers in C++
- You can use a string literal to initialize a character array or a string variable. For the array, this is preferable to a sequence of individually quoted characters.
- Adjacent string literals will be merged. This is helpful in breaking up long output formats in a `printf()` statement.
- Use `char[]` as a parameter type when the argument is a character array that may be modified. Use `char*` when the argument is a literal or a string that should not be modified.
- Declare a `char` array to hold text input or for character manipulation and take precautions that the operations will not go past the last array slot. Do not attempt to store input in a `char*` variable.
- Using the brackets specifier makes `scanf()` quite versatile, allowing you to control the number of characters read as well as the character or characters that will terminate the input operation.
- A menu-driven program provides a convenient, clear user interface. To use a menu, a list of options must be displayed where each option is associated with a code. The user is instructed to select and key in a code, which then is used to control the program's actions. Having one option on the menu to either quit or do nothing is common practice. User selections should be validated.
- One or more data arrays parallel to a menu array can be used to make calculations based on a table of data. Each array in the set represents one column in the table that relates to one data property. If integer selections are used, the items in the arrays can be indexed directly using the choice value.
- The process of using a menu is much the same for all menu applications. We introduced two menu functions that automate the process, `menu()` for numeric codes and `menu_c()` for alphabetic codes. The ragged arrays used for menus should be declared as `const` if they are defined globally.

12.6.3 Sticky Points and Common Errors

These errors are based on a misunderstanding of the nature of strings.

Ampersand errors. When using `scanf()` with `%s` to read a string, the argument must be an array of characters. Do *not* use an ampersand with the array name, because all arrays are passed by address. A character pointer (a string) can be used as an argument to `scanf()`, but it first must be initialized to point at a character array. Do *not* use an ampersand with this pointer because a pointer already is an address, and `scanf()` does not want the address of an address.

String input overflow. When reading a string as input, limit the length of the input to one less than the number of bytes available in the array that will store the data. (One byte is needed for the null character that terminates the string.) C provides some input operations, such as `gets()`, that should not be used because there is no way to limit the amount of data read. If an input operation overflows the storage, it will destroy other data or crash the program.

No room for the null character. When you allocate an array to store string data, be sure to leave space for the null terminator. Remember that most of the library functions that read and copy strings store a null character at the end of the data.

A pointer cannot store character data. Be careful to distinguish between a character array and a variable of type `char*`. The first can store a whole sequence of characters; the second cannot. A `char*` variable can only be used to point at a literal or a character array that has already been created.

Subscript validation. If the choice for a menu is not validated and if it is used to index an array, then an erroneous choice will result in accessing a location outside of the array bounds. This may cause almost any kind of error, from garbage in the output to an immediate crash.

The correct string operator. Strings in memory can be considered as two-part objects. The operations used on each part are different:

- To find the size of the pointer part, use `sizeof`; for the array part use `strlen()`.
- Remember to use the `==` operator when comparing the pointer parts, but use `strcmp()` to compare the array parts. When using `strcmp()` remember to compare the result to 0 when testing for equality.
- To copy the pointer part of a string, use `=`; to copy the array part, use `strcpy()` or `strncpy()`. Make sure the destination array is long enough to hold the result.
- The following table summarizes the syntax for initializing, assigning, or copying a string:

Operation	**With a `char` Array**	**With a `char*`**
Initialization	`char word[10] = "Hello";`	`char* message = word;`

Assignment	Does not copy the letters
	Cannot do `word = "circle"`
		`char* message = "square";`
		`message = "circle";`

String copy	`strcpy( word, "Bye" );`	Error: `strcpy( message, "Thanks" );` Cannot change a string literal.
String comparison	`strcmp( word, "Bye" )`	`strcmp( message, "Bye" )`

Quoting. A string containing one character is not the same as a single character. The string also contains a null terminator. Be sure to write single quotes when you want a character and double quotes when you want a string. The double quotes tell `C` to append a null terminator. You can embed quotation marks within a string, but you must use an escape character. Also, be careful to end all strings properly, or you may find an unwanted comment in the output.

12.6.4 New and Revisited Vocabulary

These are the most important terms and concepts presented or discussed in this chapter:

string
string literal
string merging
two-part object
null terminator
null character
null string

`NULL` pointer
conversion specifier
right and left justification
string variable
string assignment
string comparison
string concatenation

parallel arrays
`C string` library
array of strings
ragged array
menu functions
buffer overflow
menu selection validation

The following keywords and `C` library functions are discussed in this chapter.

`\"` and `\0`
`char *`
`typedef`
`char []`
`printf() %s` conversion
`scanf() %ns` conversion
`scanf %n[^?]` conversion

`gets()`
`puts()`
`strlen()`
`strcmp()`
`strncmp()`
`strcpy()`
`strncpy()`
`strcat()`

`strncat()`
`strchr()`
`strrchr()`
`strstr()`
`sizeof` a string
`==` (on strings)
`=` (on strings)

The following keywords and C++ library functions are discussed in this chapter.

`string`
`==` (on strings)
`=` (on strings)
`nullptr`
`length()`
`cin >>`
`cout <<`
`getline()`
`get()`
`replace()`
`append()`
`find_first_of()`
`find_last_of()`
`find()`

12.6.5 Where to Find More Information

- Dynamically allocated memory can be used to initialize a string variable. This is explored in Chapter 16.
- Pointers are introduced in Chapter 11 and the use of pointers to process arrays is explained in Chapter 16.
- A variety of effects can be implemented using the `scanf()` brackets conversion. The interested programmer should consult a standard reference manual for a complete description.
- Chapter 8 discusses whitespace in the input stream, the problems it can cause, and how to use `scanf()` with `%c` to handle these problems.

Chapter 13

Enumerated and Structured Types

We have discussed two kinds of aggregate data objects so far: arrays and strings. This chapter introduces a third important **aggregate type**: structures[1].

Structures provide a coherent way to represent the different properties of a single object. A `struct` variable is composed of a series of slots, called **members**, each representing one property of the object. Unlike an array, in which all parts have the same type, the components of a **structure** generally are of mixed types. Also, while the elements of an array are numbered and referred to by a subscript, the parts of a structure are given individual names. We examine how structures are represented, which operations apply to them, and how to combine arrays with structures in a compound object.

Enumerations are used to create symbolic codes for collections of conditions or objects. They can be useful in a variety of situations, such as handling error conditions, where they list and name the possible types of outcomes.

13.1 Enumerated Types

The use of `#define` to give a symbolic name to a constant is familiar by now. It is a simple way to define a symbolic name for an isolated constant such as π or the force of gravity. Sometimes, though, we need to define a set of related symbols, such as codes for the various kinds of errors that a program can encounter. We wish to give the codes individual names and values and declare that they form a set of related items. The `#define` command lets us name values individually but gives us no way to associate them into a group of codes whose meaning is distinct from all other defined symbols. For example, suppose we wanted to have symbolic names for the possible ways that a data item could be illegal. We could define the following four symbolic constants:

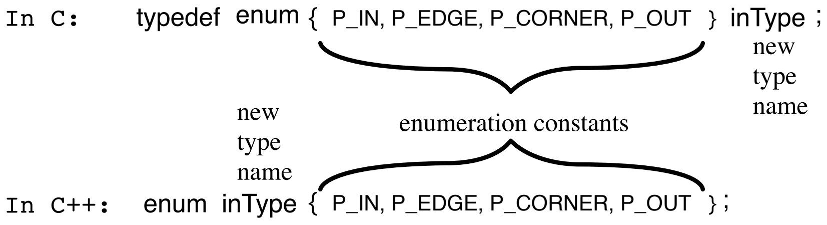
```
#define DATA_OK  0
#define TOO_SMALL  1
#define TOO_BIG  2
#define NO_INPUT -1
```

Enumerated types were invented to address this shortcoming. They allow us to define a type name that groups together a set of related symbolic codes.

13.1.1 Enumerations

The character data type really is an enumeration. The ASCII code maps the literal forms of the characters to corresponding integer values. This particular enumeration is built into the C language, so you do not see its construction. But it is possible to create new types using the **enumeration specification**, the keyword `enum` followed by a bracketed list of symbols called **enumeration constants**. Enumerations normally are defined within a `typedef` declaration, as shown in Figure 13.1.

[1] Union data types are similar to structures with more than one possible configuration of members. They are not covered in this text because their applications are better covered by polymorphic types in modern languages. Applications that require such types are better programmed in C++ or Java.


In C: typedef enum { P_IN, P_EDGE, P_CORNER, P_OUT } inType ;
new type name
enumeration constants
new type name
In C++: enum inType { P_IN, P_EDGE, P_CORNER, P_OUT } ;

The underlying values of P_IN...P_OUT are 0, 1, 2, 3 respectively.

Figure 13.1. The form of an enum declaration in C and C++.

Although the enum and #define mechanisms produce similar results, using an enum declaration is superior to using a set of #define commands because we can declare that the enumeration codes listed form the complete set of relevant codes for a given context. An enumerated type can, therefore, define the set of valid values a variable may have or a function may return. Using enum rather than #define improves readability with no cost in efficiency.

The enumeration symbols will become names for constants, much as the symbol in a #define becomes a name for a constant. By default, the first symbol in the enumeration is given the underlying value 0; succeeding symbols are given integer values in ascending numeric order, unless the sequence is reset by an explicit assignment. Such an assignment is illustrated by the third declaration in Figure 13.2. An explicit initializer resets the sequence, which continues from the new number, so it is possible to leave gaps in the numbering or have two constants represented by the same code, which normally is undesirable. Therefore, the programmer must be careful when assigning specific representations to enumeration constants.

Inside a C program, enumeration values are treated as integers in all ways, as is true for characters. The compiler makes no distinction between a value of an enumeration type and a value of type int. Since enumeration constants are translated into integers, all the integer operators can be used with enumeration values. Although most integer operators make no sense with enumeration constants, increment and decrement operators sometimes are used, as they are with type char, to compute the next value in the code-series. However, unlike char, C provides no way to automatically read or print enumeration values; they must be input and output as integers. This complication leads to extra work to devise special input and output routines for values of an enumeration type. The extra hassle sometimes discourages the programmer from using an enum type. However, the added clarity in a program normally is worth the effort, and in truth, the values of many such types never actually are input or output, but merely used within the program to send information back and forth between functions.

In a C++, program, enumerated types are distinct from integers, but can be converted to and from integers by casting.

Why use enumerations? We use enumerations (rather than integer constants) to distinguish each kind of code from ordinary integers and other codes. This enables us to write programs that are less cryptic. Someone reading the program sees a symbol that conveys its meaning better than an integer that encodes the meaning. The reader understands the program more easily because the use of enum clarifies which symbols are related to each other. By using an enum constant rather than its associated integer, we make it evident that the object is

```
typedef enum { P_IN, P_SIDE, P_CORNER, P_OUT } inType;
typedef enum { NO_ROOT, ROOT_OK } statusT;
typedef enum { DATA_OK, TOO_SMALL, TOO_BIG, NO_INPUT=-1 } errorT;

enum inType { P_IN, P_SIDE, P_CORNER, P_OUT };
enum statusT { NO_ROOT, ROOT_OK };
enum errorT { DATA_OK, TOO_SMALL, TOO_BIG, NO_INPUT=-1 };
```

Figure 13.2. Four enumerations in C and again in C++.

We declare an enumeration variable and use it to store an error code after testing an input value.

```
double x;
errorT status;

scanf( "%lg", &x );
if (x < 0) status = TOO_SMALL;
else status = DATA_OK;
```

Figure 13.3. Using an enumeration to name errors.

a code, not a number.

Notes on Figure 13.2. Four enumerations. This figure declares three enumerated types that give names to conditions that are somewhat complex to characterize but easy to understand once they are named. These types will be used in various programs in the remainder of the text.

Type* `in_type`*: answer codes. The first `typedef` declares a set of codes that will be used in an application in this Chapter. This program processes a point and a rectangle in the xy plane, determining where the point is located with respect to the rectangle. One function does the analysis and returns an answer of type `in_type` to the main program, which then interacts with the user. The code `P_IN` is used if the point is inside the rectangle; the codes `P_SIDE, P_CORNER`, and `P_OUT`, respectively, mean that the point is on a side, on a corner, or outside of the rectangle.

Type* `status_t`*: status codes. The second declaration in Figure 13.2 is an enumeration of codes used in a program that finds the roots of an equation (Figure 13.7). In this program, the main program prompts the user for equation parameters, then calls a function to do the actual work. The function returns the root (if found) and a success-or-failure code, `ROOT_OK` or `NO_ROOT`. The descriptive nature of these symbols makes the program logic clearer.

Type* `error_t`*: error codes. The third declaration in Figure 13.2 is an enumeration of **error codes**, which can be used to simplify the management of input errors. The symbols `DATA_OK`, `TOO_SMALL`, and `TOO_BIG` carry the underlying integer values 0, 1, and 2, respectively. The last symbol, `NO_INPUT`, is followed by an assignment, so the symbol `NO_INPUT` will be represented by the code -1 rather than 3. Figure 13.3 illustrates a code fragment that applies this enumerated type. These statements test for an input error and set the value of a variable to indicate whether it was detected. Further input error processing is described in Chapter 14.

13.1.2 Printing Enumeration Codes

Since enumeration constants are represented by integers in `C`, we could print them as integers. However, the purpose of using an enumeration is to give meaningful symbolic names to status codes, so we certainly do not want to see a cryptic integer code in the output. Rather, we want to see an English word.

Symbolic output is achieved easily using a ragged array. An array of strings is declared that is parallel to the enumeration. Each word in the array is the English version of the corresponding symbol in the enumeration. We use the symbol (an integer) to subscript the ragged array and select the corresponding string, which then is used in the output. The array of strings is defined in the same program scope as the enumeration, so that any function that uses the enumeration has access to the output strings. Since enumerations usually are defined globally, the string array also is global. An important precaution with any global array is to begin the declaration with the keyword `const` so that no part of the program code can change it accidentally. As an example, consider an enumeration defined in Chapter 8:

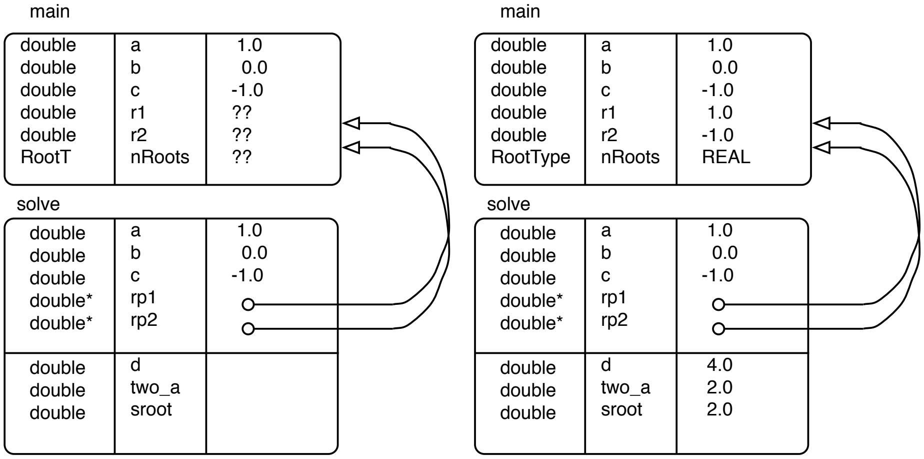
```
typedef enum { P_IN, P_SIDE, P_CORNER, P_OUT } in_type;
```

We can define output strings thus:

```
const char* in_labels[] = {  "inside", "on a side of", "on a corner of", "outside" };
```

To show how to use this array of strings, assume we have a variable, `position`, that contains a value of type `in_type`. We write the following lines to insert the English equivalent of the value of `position` into the middle of a sentence and display the result on the screen:

We diagram the storage allocated for the quadratic root program in Figures 13.5 and 13.6. The diagram on the left shows the contents of memory just after calling the `solve()` function and before its code is executed. The diagram on the right shows the situation during the function return process, just before the function's storage disappears.



main

double	a	1.0
double	b	0.0
double	c	-1.0
double	r1	??
double	r2	??
RootT	nRoots	??

solve

double	a	1.0
double	b	0.0
double	c	-1.0
double*	rp1	
double*	rp2	
double	d	
double	two_a	
double	sroot	

main

double	a	1.0
double	b	0.0
double	c	-1.0
double	r1	1.0
double	r2	-1.0
RootType	nRoots	REAL

solve

double	a	1.0
double	b	0.0
double	c	-1.0
double*	rp1	
double*	rp2	
double	d	4.0
double	two_a	2.0
double	sroot	2.0

Figure 13.4. Memory for the quadratic root program.

```
    in_type position;
    ...
    printf( " The point is %s the square.\n", in_labels[position] );
```

The output might look like this

```
  The point is on a corner of the square.
```

13.1.3 Returning Multiple Function Results

The program in Figures 13.5 through 13.7 illustrate a typical use of an enumerated type and the use of call by address to return multiple results from a function. The enumerated type helps to make this code very straightforward and readable.

The type definition and main program (in C, Figure 13.5 then in C++, Figure 13.6, does only user interaction, input and output. The equation is solved by the function in Figure 13.7 which must then return one, two, or three results, depending on the coefficients of the equation. This code is the same for both C and C++.

```
#include <stdio.h>
#include <math.h>
#include <stdbool.h>
typedef enum  ZERO, NONE, LINEAR, SINGLE, REAL, COMPLEX  RootT;
RootT solve( double a, double b, double c, double* rp1, double* rp2 );
// -------------------------------------------------------------
int main( void )
{
    double a, b, c;                 // Coefficients of the equation.
    double r1, r2;                  // Roots of the equation.
    RooT nRoots;                    // How many roots the equation has.
    puts( "\ Find the roots of a*x^2 + b*x + c = 0" );
    printf( " Enter the values of a, b, and c: " );
    scanf( "%lg%lg%lg", &a, &b, &c );
    printf( "\n The equation is %.3g *x^2 + %.3g *x + %.3g = 0\n", a, b, c );
    nRoots = solve( a, b, c, &r1, &r2 );
    switch (nRoots) {
      case ZERO:
           printf( "\n Degenerate equation -- a = b = c = 0.\n\n" );
           break;
      case NONE:
           printf( "\n Degenerate equation -- no roots\n\n" );
           break;
      case LINEAR:
           printf( "\n Linear equation -- root at -c/b = %.3g\n\n", r1 );
           break;
      case SINGLE:
           printf( "\n Single real root at x = %.3g \n\n", r1 );
           break;
      case REAL:
           printf( "\n The real roots are x = %.3g and %.3g \n\n", r1, r2 );
           break;
      case COMPLEX:
           printf( "\n The complex roots are x = %.3g + %.3g i and "
                   "x = %.3g - %.3g i \n\n", r1, r2, r1, r2 );
    }
    return 0;
}
```

Figure 13.5. Solving a quadratic equation in C.

Figure 13.6. Solving a quadratic equation in C++.

Notes on Figures 13.5 and 13.6: Using the quadratic formula to solve an equation.

First box, Figure 13.5 and lines 6 of Figure 13.6: **status code enumeration** *of cases.* We define an enumerated type to represent the five possible ways in which a quadratic equation can have or not have roots.

1. `NONE` means the equation is degenerate because the coefficients are all zero.
2. `NONE` means the equation is inconsistent; it has a nonzero constant term but no variable terms.
3. `LINEAR` means the equation is linear because the coefficient of x^2 is 0.
4. `SINGLE` means the two real roots of this equation are the same; if graphed, the function would be tangent to the x axis at the root value.
5. `REAL` means there are two real roots; if graphed, the function would cross the x axis twice.
6. `COMPLEX` means there are two complex roots; if graphed, the function would not touch or cross the x axis.

Second box, Figure 13.5 and lines 7 and 22 of Figure 13.6: calling the `solve()` ***function.*** This function call takes three arguments into the function (the three coefficients of the equation), passing them by value. It also supplies two addresses in which to store the roots of the equation, should they exist. The value returned directly by the function will be one of the status codes from the enumeration in the first box. In Figure 13.4, note that the values of the first three arguments have been copied into the memory area of the function, but addresses (pointers) are stored in the last two parameters.

Third box, Figure 13.5 and lines 23–42 of Figure 13.6: interpreting the results. The status code returned by the function is stored in `nRoots`. We use a `switch` statement to test the code and select the appropriate format for displaying the answers. It is quite common to use a `switch` statement to process an enumerated-type result because it is a simple and straightforward way to handle a complex set of possibilities. It is not necessary to use a default case, since all of the enumerated type's constants have been covered. This programming technique also has the advantage that it separates the user interaction from the algorithm that computes the answers, making both easier to read and write.

Output.

```
Find the roots of a*x^2 + b*x + c = 0
Enter the values of a, b, and c: 1 0 -1
The equation is  1 *x^x + 0 *x + -1 = 0

The real roots are x = 1 and -1
```

Notes on Figures 13.7 and 13.4. The quadratic root function.

This computational function is the same for C and C++.

First box: A zero test.

- We use `double` values for the coefficients of the equation. To make the program more robust and useful in various circumstances, we use an epsilon test for all comparisons. We arbitrarily choose an epsilon value of 10^{-100}, which is small enough to be virtually zero but large enough to ensure that using it as a divisor will not cause floating point overflow.
- The `iszero()` function uses the `EPS` value to compare its argument to 0. We write the code as a separate function to remove repetitive clutter from the `solve()` function, making it shorter and clearer.
- This function returns a `bool` result, which can be used directly in an `if` statement or a logical expression.

Second box: degenerate cases.

- We test for coefficients that do not represent quadratic equations and return the appropriate code from the enumerated type. Some sort of test for zero is necessary to avoid the possibility of division by zero or near zero in the third and fourth boxes. We use a tolerance of 10^{-100}.
- By using the `if...return` control structure, we eliminate the need for `else` clauses and nested conditionals. The added logic needed to support a single return statement would be a considerable distraction from the simplicity of this solution.
- Error output:

```
Find the roots of a*x^2 + b*x + c = 0
```

This function is called from the quadratic root program in Figures 13.5 and 13.6. One result is returned by a `return` statement; the other two (if they exist) are returned through pointer parameters.

```
#define EPS  1e-100
bool iszero( double x ) { return fabs( x ) < EPS; }
```

```
RootT    // Return value is the enum constant for number and type of roots.
solve( double a, double b, double c, double * rp1, double * rp2 )
{
    double d, two_a, sroot;        // Working storage.

    if (iszero( a ) && iszero( b )) {  // Degenerate cases.
        if (iszero( c )) return ZERO;
        else return NONE;
    } if (iszero( a )) {
        *rp1 = -c / b;
        return LINEAR;
    }

    two_a = 2 * a;
    d = b * b - 4 * a * c;        // discriminant of equation.
    if (iszero( d )) {            // There is only one root.
        *rp1 = -b / two_a;
        return SINGLE;
    }

    sroot = sqrt( fabs( d ) );    // fabs is floating point absolute value.
    if (d > EPS) {                // There are 2 real roots.
        *rp1 = -b / two_a + sroot / two_a;
        *rp2 = -b / two_a - sroot / two_a;
        return REAL;
    }
    else {                        // There are 2 complex roots.
        *rp1 = -b / two_a;
        *rp2 = sroot / two_a;
        return COMPLEX;
    }
}
```

Figure 13.7. The quadratic root function for both C and C++.

```
 Enter the values of a, b, and c: 0 0 0
 The equation is  0 *x^x + 0 *x + 0 = 0

 Degenerate equation -- a = b = c = 0.
 -------------------------------------------------------
 Find the roots of a*x^2 + b*x + c = 0
 Enter the values of a, b, and c: 0 0 1
 The equation is  0 *x^x + 0 *x + 1 = 0

 Degenerate equation -- no roots
 -------------------------------------------------------
 Find the roots of a*x^2 + b*x + c = 0
 Enter the values of a, b, and c: 0 3 1
 The equation is  0 *x^x + 3 *x + 1 = 0

 Linear equation -- root at -c/b = -0.333
-------------------------------------------------------
```

Third box: A single root.

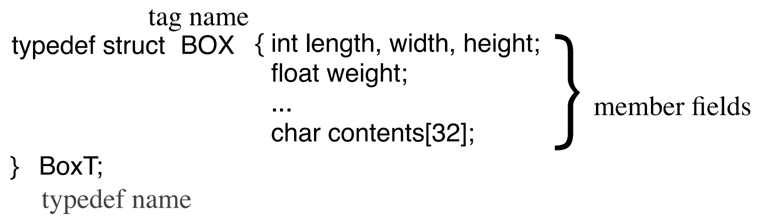
- We compute and store the values of two_a and d because these subexpressions are used several times in this box and the next. Factoring out these computations increases run-time efficiency and shortens the code.
- If the equation is quadratic, it can have two separate roots, real or complex, or one repeated real root. We look at the discriminant of the equation, d, to identify how many roots are present. If d equals 0, the equation has only one distinct root, which we then compute and return through the first pointer parameter.
- The function return value tells the main program that only one of the pointer parameters has been given a value (the other is unused).
- Single root output:

```
Find the roots of a*x^2 + b*x + c = 0
Enter the values of a, b, and c: 2 4 2
The equation is  2 *x^x + 4 *x + 2 = 0

Single real root at x = -1
```

Fourth box: Two different roots.

- We compute sroot once and use the stored value to compute the roots in each of the two remaining cases.
- In both cases, two double values are returned to main() by storing them indirectly through the pointer parameters. The function return value tells main() how to interpret these two numbers.
- The REAL case is illustrated in Figure 13.4 and the corresponding output is shown in the notes for the main program. Note that the values of r1 and r2 in main() have been changed by storing numbers indirectly through the two pointer parameters.
- Output with two roots, real or complex:

```
Find the roots of a*x^2 + b*x + c = 0
Enter the values of a, b, and c: 1 0 -1
The equation is  1 *x^x + 0 *x + -1 = 0

The real roots are x = 1 and -1
-------------------------------------------------------

Find the roots of a*x^2 + b*x + c = 0
Enter the values of a, b, and c: 1 0 1
The equation is  1 *x^x + 0 *x + 1 = 0

The complex roots are x = -0 + 1 i and x = -0 - 1 i
```



```
typedef struct  BOX  { int length, width, height;
                       float weight;
                       ...
                       char contents[32];
}  BoxT;
```

Figure 13.8. Modern syntax for a C `struct` declaration.

13.2 Structures in C

This section presents structures in C and explains the syntax used to do operations on structures. The following section builds on these explanations and shows how to use structures and classes in C++

A structure type specification starts with the keyword `struct`, followed by an optional identifier called the **tag name**, followed by the member declarations enclosed in curly brackets. (Members also may be called *fields* or *components*.) All this normally is written as part of a `typedef` declaration, which gives another name, the **typedef name**, to the type. The members can be almost any combination of data types, as in Figure 13.8. Members can be arrays or other types of structures. However, a structure cannot contain a member of the type it is defining (the type of a part cannot be the same as the type of the whole).

New types. A new type name can be defined using `typedef` and `struct`, (See Figure 13.8.) and may be used like a built-in type name anywhere a built-in type name may be used. Important applications are to

1. Declare a structured variable.
2. Create an array of structures.
3. Declare the type of a function parameter.
4. Declare the type of a function result.

Before the `typedef` was introduced into C, the tag name was the only way to refer to a structured type. Using tag names is syntactically awkward because the keyword `struct` is part of the type name and must be used every time the tag name is used. For example, a type definition and variable declaration might look like this:

```
struct NAME { char first[16]; char last[16]; } boy;
```

or they could be given separately, like this:

```
struct NAME { char first[16]; char last[16]; };
struct NAME boy;
```

Initially, this was addressed in C by introducing the `typedef` declaration. In C++, however, the problem was solved properly. Now, with either a `struct` or a `class`, the name following the keyword is the name of the type.

Today, tag names are not used as much in C because `typedef` is less error prone and serves most of the purposes of tag names. A tag name is optional when a `typedef` declaration is given and the `typedef` name is used. Tag names have two remaining important uses. If one is given, some on-line debuggers provide more informative feedback. They also are useful for defining recursive types such as trees and linked lists.[2] When present, the tag name should be related to the `typedef` name. Naming style has varied; the most recent seems to be to write tag names in upper case and `typedef` names in lower case. We supply tag names with no further comment for the types defined in this chapter.

The result of a `typedef struct` declaration is a "pattern" for creating future variables; it is not an object and does not contain any data. Therefore, you do not include any initializers in the member declarations.

[2] In such types, one member is a pointer to an object of the `struct` type currently being defined. These topics are beyond the scope of this text.

A struct type specification creates a type that can be used to declare variables. The typedef declaration names this new type.

```
typedef struct LUMBER {       // Type for a piece of lumber.
    char wood[16];            // Type of wood.
    short int height;         // In inches.
    short int width;          // In inches.
    short int length;         // In feet.
    float price;              // Per board, in dollars.
    int quantity;             // Number of items in stock.
} LumberT;
```

The members of LumberT will be laid out in memory in this order. Depending on your machine architecture, padding bytes might be inserted between member fields.

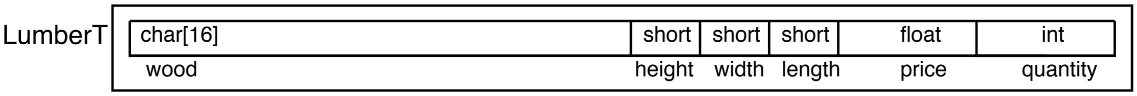


Figure 13.9. A struct type declaration in C.

Structure members. Member declarations are like variable declarations: Each consists of a type, followed by a list of member names ending with a semicolon. Any previously defined type can be used. For example, a structure listing the properties of a piece of lumber is defined in Figure 13.9. We represent the dimensions of the board as small integers (a 2 × 4 × 8 board is cut at the sawmill with a cross section of approximately 2 × 4 inches and a length of 8 feet). We use the types float to represent the price of the lumber and int to represent the quantity of this lumber in stock. Figure 13.9 also shows a memory diagram of the structure created by this declaration.

The **member names** permit us to refer to individual parts of the structure. Each name should be unique and convey the purpose of its member but do so without being overly wordy. (This is good advice for any variable name.) A member name is used as one part of a longer compound name; it needs to make sense in context but need not be complete enough to make sense if it stood alone.

A component **selection expression** consists of an object name followed by one or more member names, all separated by the dot operator. Because such expressions can get lengthy, it is a good idea to give each member a brief name.

We use the type declared in Figure 13.9.

```
LumberT sale;
LumberT plank = { "white oak", 1, 6, 8, 5.80, 158 };
int bytes = sizeof (LumberT);
```

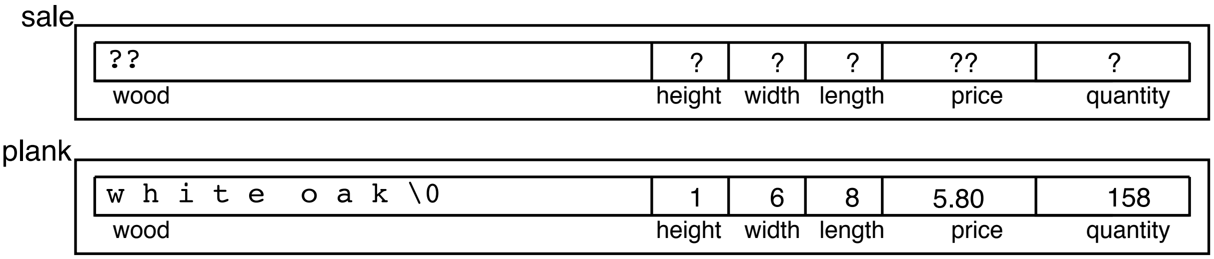


Figure 13.10. Declaring and initializing a structure.

Declaring and initializing a C structure. We can declare a structured variable and, optionally, initialize it in the declaration. The two declarations in Figure 13.10 use the structured type declared in Figure 13.9 to declare variables named `sale` and `plank`. The variable `sale` is uninitialized, so the fields will contain unknown data. The variable `plank` is initialized. The **structured variable initializer** is much like that for an array. It consists of curly brackets enclosing one data value, of the appropriate type, for each member of the structure. These are separated by commas and must be placed in the same order as the member declarations. The arrangement of the values in `plank` is diagrammed in the figure. Like an array, if there are too few initializers, any remaining member will be filled with 0 bytes.

13.3 Operations on Structures

ISO C supports a variety of **operations on structures**;[3] Standard ISO C compilers permit the following operations:

1. Find the size of the structure.
2. Set a pointer's value to the address of a structure.
3. Access one member of a structure.
4. Use assignment to copy the contents of a structure.
5. Return a structure as the result of a function.
6. Pass a structure as an argument to a function, by value or address.
7. Include structures in larger compound objects, such as an array of structures.
8. Access one element in an array of structures.
9. Access one member of one structure in an array of structures.

One important and basic operation is not supported in C: comparison of two structures. When comparison is necessary, it must be implemented as a programmer-defined function. The remainder of this section isolates and explains each of these operations. Figures 13.14, 13.15, 13.16, and 13.18 contain function definitions that are part of the complete program in Figures 13.20 and 13.21.

13.3.1 Structure Operations

The size of a structure. C was designed to be used on many kinds of computers, with various hardware characteristics. Since the language was introduced in 1971, hardware has changed drastically and is still changing. The language standard specifies how a program should behave, but not how that program should be represented in machine language. The C standard states that the members of a structure must be stored in order, but it permits extra bytes, or *padding* to be inserted between members. Thus, some structures occupy more total bytes than the sum of the sizes of their members. When padding is used, the extra bytes are inserted by the compiler into the structure to force each structure member to start on a memory address that is a multiple of 2 (for small computers) or 4 (most modern computers). This process is called *alignment* and it is done to match hardware requirements or to improve efficiency.

Padding is only used after small or irregular-sized members. It is not an issue with types `double`, `float`, `long`, or `int`, because these types already meet or exceed all hardware alignment requirements. Single characters and odd-sized strings, however, will likely be followed by padding. In the `LumberT` structure, the first member is a `char` array of length 11. It will be padded by 1 byte in any modern compiler. The next three members are type `short` integers. Some compilers might pad each one to four bytes, and some compilers will not add padding at all. Most current compilers will insert two bytes of padding after the third `short int`, because the `float` that follows it must be aligned to an address that is a multiple of 4. Thus, the size of the structure pictured in Figure 13.10 could be as little as 24 bytes (if `sizeof(int) = 2`) or as great as 32 bytes, (if `sizeof(int) = 4` and all possible padding was added). In the `gcc` compiler, it is 28 bytes.

[3] Older, pre-ANSI C compilers may not support some operations, particularly assignment. Therefore, what you are permitted to do with a structure depends on the age of your compiler.

```
p = & plank;
```

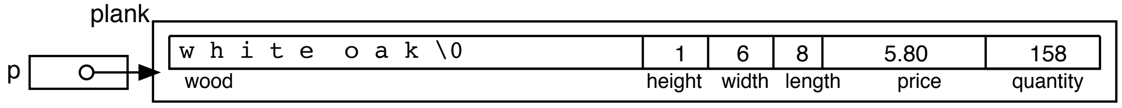


Figure 13.11. Set and use a pointer to a structure.

In most situations, the programmer does not need to know whether padding is used by the compiler. Sometimes, however, a program must manage its own memory, and needs to know how large a structure is in order to allocate memory space for it[4]. `sizeof` operator is used for this purpose, as shown in the first line in Figure 13.10.

Set a pointer to a structure. We can declare a pointer and set its value to the address of a `struct`. In Figure 13.11, the address of the structure named `plank`, defined in Figure 13.10, is stored in the pointer variable `p`. The address used is the address of the first member of the structure, but the pointer gives access to the entire structure, as illustrated.

Access one member of a structure. The members of a structure may be accessed either directly or through a pointer. The **dot** (period) **operator** is used for direct access. We write the variable name followed by a dot, followed by the name of the desired member. For example, to access the `length` member of the variable `plank`, we write `plank.length`. Once referenced, the member may be used anywhere a simple variable of that type is allowed.

The **arrow operator** is used to access the members of a structure through a pointer. We write the pointer name followed by `->` (arrow) and a member name. For example, since the pointer `p` is pointing at `plank`, we can use `p` to access one member of `plank` by writing `p->price`. This is illustrated by the last line in Figure 13.11. We use both access operators in Figure 13.12 to print two members of `plank`. The output from these two print statements is

```
Plank price is $5.80
Planks in stock: 158
```

Structure assignment. ISO compilers let us copy the entire contents of a structure in one assignment statement. A structure assignment copies all of the values of the members of the structure on the right into the corresponding members of the structure variable on the left (which must be the same type).[5] Figure 13.13 gives an example.

13.3.2 Using Structures with Functions

Three functions are given in this section to illustrate how structures can be passed into and out of functions. Calls on these functions are incorporated into the program in Figures 13.20 and 13.21.

Returning a structure from a function. The same mechanism that allows us to do structure assignment lets us return a structure as the result of a function. For example, in Figure 13.14, we declare a function, `read_lumber()`, that reads the data for a piece of lumber into the members of a local structured variable. In

```
printf( " %s price is $%.2f\n", plank.wood, plank.price );   // direct access
printf( " %s in stock:  %i\n", p->wood, p->quantity );       // indirect access
```

Figure 13.12. Access one member of a structure.

[4] This is explained in Chapter 16.

[5] This actually is a “shallow” copy. If one of the members is a pointer to an object, assignment makes a copy of the address but not the object it references. Then two objects refer to a common part and any change in that part is “seen” in both copies! Deep copies, which duplicate both pointer and referent, are beyond the scope of this text.

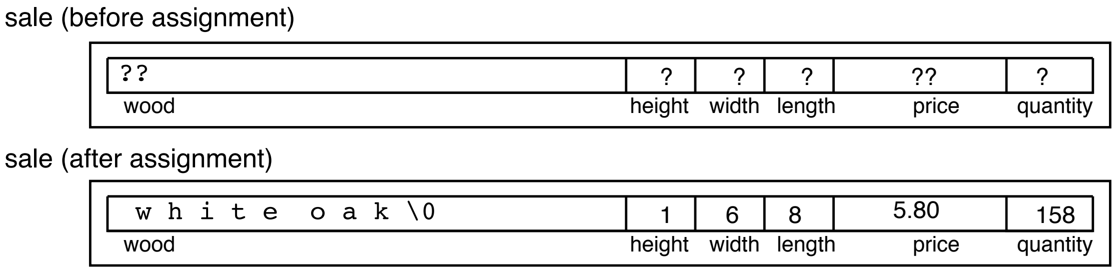


Figure 13.13. Structure assignment.

these `scanf()` statements, we do not need parentheses around the entire member name. The dot operator has higher precedence than the address operator, so we get the address of the desired member. When all input has been read, we return a copy of the contents of the local structure as the function's value[6].

Guidance: This is yet another way to return multiple values from a function, but should be avoided if the structure has more than a few members because of the large amount of copying involved. For large structures, call by address should be used in stead.

Call by value with a structure. We can pass a structure as an argument to a function. When we use call by value to do this, all the members of the caller's argument are copied into the members of the function's parameter variable, just as if each were a separate variable. Thus, the technique is used primarily for small structures and call by const address is preferable for structures with more than a few members.

In Figure 13.15, we declare a function, `print_lumber()` with a structure parameter. It formats and prints

[6] Note that we are returning a copy of the local variable, not a pointer to it.

```
LumberT read_lumber( void )
{
     LumberT board;
     printf( " Reading a new stock item.  Enter the three dimensions of a board: " );
     scanf( "%hi%hi%hi", &board.height, &board.width, &board.length );
     printf( " Enter the kind of wood: " );
     scanf( "%15[^\n]", board.wood );
     printf( " Enter the price: $" );
     scanf( "%g", &board.price );
     printf( " Enter the quantity in stock: " );
     scanf( "%i", &board.quantity );
     return board;
}
```

Figure 13.14. Returning a structure from a function.

```
void print_lumber( LumberT b )
{
    printf( " %s  %hi\" x %hi\" x %hi feet long -> %i in stock at $%.2f\n",
            b.wood, b.height, b.width, b.length, b.quantity, b.price );
}
```

Figure 13.15. Call by value with a structure.

```
void sell_lumber( int sold, LumberT* board )
{
    if (board->quantity >= sold) {
        board->quantity -= sold;
        printf( " OK, sold %i\n", sold %i %s\n", sold, board->wood );
        print_lumber( *board );
    }
    else printf( " Error: cannot sell %i %s boards; have only %i.\n",
                   sold, board->wood, board->quantity );
}
```

Figure 13.16. Call by address with a structure.

the data in the structure. Because the parameter is a structure (not a pointer), we use the dot operator to access the members. Calls on `read_lumber()` and `print_lumber()` might look like this.

```
sale = read_lumber();
print_lumber( sale );
```

The input would be read into a local variable in the `read_lumber()` function, then copied into `sale` when the function returns. It would be copied a second time into parameter `b` in `print_lumber()`. Calls on `read_lumber()` and `print_lumber()` using pointers might look like this:

```
*p = read_lumber(); />// Store a structured return value.
print_lumber( *p ); /> // Echo-print stock[3].
```

The output produced by this interaction might be

```
Reading a new stock item.
Enter the three dimensions of a board: 2 3 6
Enter the type of wood: fir
Enter the price: $4.23
Enter the quantity in stock: 16
fir       2" x 3" x 6 feet  $4.23  stock: 16
```

Call by address with a structure. We can pass a pointer to a structure, or the address of a structure, as an argument to a function. This, **call by address**, gives us more functionality than call by value. In Figure 13.16, we declare a function named `sell_lumber()` that takes an integer (the item number) and a structure pointer as its parameters and checks the inventory of that item. If the supply is adequate, the quantity on hand is reduced by the number sold; otherwise an error comment is printed. The following sample calls illustrate two ways to call this function:

```
sell_lumber( p, 200);            // A pointer argument.
sell_lumber( &stock[1], 2 );
```

The output would look like this:

```
Error: cannot sell 200 boards (only have 158).
OK, sold 2
fir       2" x 3" x 6 feet  $4.23  in stock: 14
```

On the first line of `sell_lumber()`, we use the parameter (a pointer) with the arrow operator to access a member of the structure. Within the code is a call on `print_lumber()` to show the reduced quantity on hand after a sale. Note that the pointer parameter of `sell_lumber()` is the wrong type for `print_lumber()`, because the argument for `print_lumber()` must be an item, not a pointer. We overcome this incompatibility by using a dereference operator in the call to `print_lumber()`. The result of the dereference is a structured value.

Value vs. address parameters. The three functions just discussed, as well as the `boards_equal()` function in Figure 13.18 have been implemented using call by value wherever possible; it was necessary to use a pointer parameter (call by address) for `sell_lumber()`, because the function updates its parameter to return information to the caller. The prototypes are

```
LumberT read_lumber( void );
void print_lumber( LumberT );
void sell_lumber( int, LumberT* );
bool boards_equal( LumberT, LumberT );
```

Pointer parameters also could have been used in `print_lumber()`, `read_lumber()`, and `boards_equal()`. This would have the advantage of reducing the amount of copying that must happen at run time because only a pointer, not an entire structure, must be copied into the function's parameter. The time and space needed for copying can be significant for large structures. The same is true of the time consumed by returning a structured result from a function like `read_lumber()`. If the structure has many members or contains an array of substantial size as one member, it usually is better to use call by address to send the information to a function and to return information.

In any case, call-by-address requires extra care because it does not automatically isolate the caller from the subprogram. Thus, the subprogram can change the value stored in the caller's variable. In functions like `print_lumber()` and `boards_equal()` that use, but do not modify, their parameters, a `const` should be added to the parameter type to prevent accidental mistakes that are hard to track and correct. The revised prototypes for a large structure in this program would be

```
void read_lumber( LumberT* );
void print_lumber( const LumberT* );
void sell_lumber( int, LumberT* );
bool boards_equal( const LumberT*, const LumberT* );
```

One disadvantage of using a pointer parameter is that every reference to every member of the structure is *indirect*; first the reference address must be fetched from the parameter, then it must be used to access the memory location of the needed member. If members of the structure are used many times in the function, these extra references can substantially slow down execution, and it is better to use call by value.

Finally, when call by value is used, a copy is made. Sometimes the copy plays an important part in the program. For example, when modifying the data in one record of a database, it is good engineering practice to permit the process to be aborted midway. To do this, all modifications are made to a local *copy* of the data, and the original remains unchanged until the modification process is finished and the user confirms that the result is acceptable. At this time, the changes are *committed*, and the revised data record is returned to the caller and copied into the location of the original. In actual practice, this is a very important technique for maintaining a database of any sort, with or without an underlying database engine.

13.3.3 Arrays of Structures

It is possible to include one aggregate type within another. In the definition of the `struct` for `LumberT`, we include a character array. A `struct` type also may be the base type of an array. Thus, we can declare and, optionally, initialize an **array of structures**. An initializer for an array of structures is enclosed in curly brackets and contains one bracketed set of values for each item in the array, separated by commas.[7] An example and its diagram are shown in Figure 13.17, where we declare and initialize an array of `LumberT` structures.

Accessing one structure in an array of structures. The name of the array and a subscript are needed to access one element in an array of structures, shown in the following code. The first line assigns a structured value to the last item of an array, the second calls `print_lumber()` to display the result. To send one structure from the array, by address, to a function that has a structure pointer parameter, the argument needs both a subscript and an ampersand, as shown by the call on `sell_lumber()` on the third line.

```
stock[3] = read_lumber();       // Read into one element.
print_lumber( stock[3] );       // Print one element.
sell_lumber( &stock[3], 2 );    // Call by address.
```

[7] In addition to the commas between members, it is legal to have a comma after the last one. This sometimes makes it easier to create large data tables with a word processor, without having to remember to remove the comma from the last one.

```
LumberT stock[5] = { { "spruce", 2, 4, 12, 8.20, 27 },
                     { "pine", 2, 3, 10, 5.45, 11 },
                     { "pine", 2, 3, 8, 4.20, 35 },
                     { "" // Remaining members will be set to 0 }
                   };
```

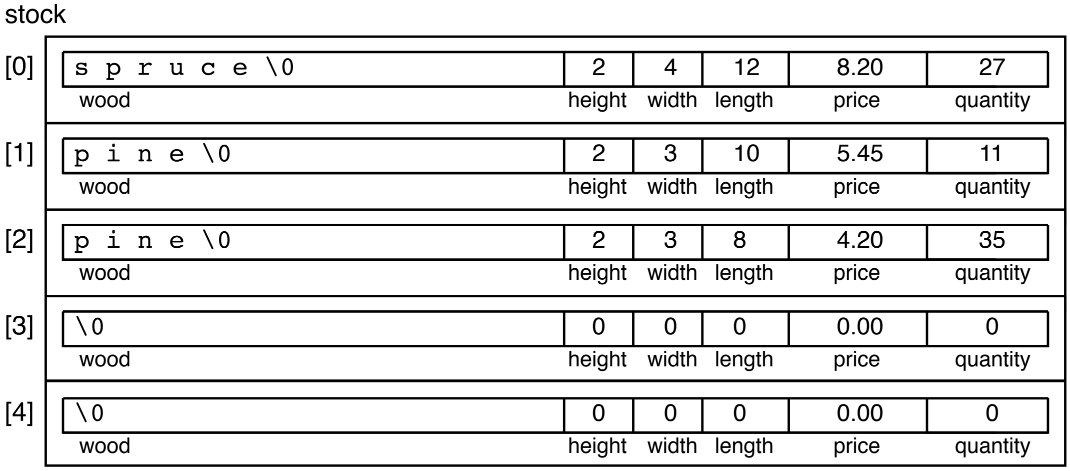


Figure 13.17. An array of structures.

A sample of the interaction and output is:

```
Reading a new stock item.
Enter the three dimensions of a board: 1 4 10
Enter the type of wood: walnut
Enter the price: $29.50
Enter the quantity in stock: 10
walnut     1" x 4" x 10 feet  $29.50  stock: 10
OK, sold 3
walnut     1" x 4" x 10 feet  $29.50  stock: 7
```

Accessing a member of one structure in an array. We also can access a single member of a structure element. Both a subscript and a member name must be used to access an individual part of one structure in an array of structures. Again, no parentheses are needed because of the precedence of the subscript and dot operators. Write the subscript first, because the overall object is an array. The result of applying the subscript is a structure to which the dot operator is applied. The final result is one member of the structure. For example, to print the `price` of the second piece of lumber in the `stock` array, we would write this:

```
printf( "Second stock item: $%.2f\n", stock[1].price );
```

This statement's output is

```
Second stock item: $5.45
```

Arrays of structures vs. parallel arrays. In Section 10.3, we used a set of **parallel arrays** to implement a table. In this representation, each array represented one column of the table, and each array position represented one row. An array of structures can represent the same data: Each structure is a row of the table and the group of structure members with the same name represents one column.

A modern approach to representation would favor grouping all the columns into a structure and using an array of structures. This combines related data into a coherent whole that can be passed to functions as a single argument. The method is convenient and probably helps avoid errors. The argument in favor of using parallel arrays is that the code is simpler. The column's array name and a subscript are enough to access any property.

Type bool was defined in Figure 13.2. We use it here as the return type of a function that compares two structured variables for equality.

```
bool boards_equal( LumberT board1, LumberT board2 )
{
    return (strcmp(board1.wood, board2.wood) == 0) &&
           (board1.height   == board2.height) &&
           (board1.width    == board2.width)  &&
           (board1.length   == board2.length) &&
           (board1.price    == board2.price)  &&
           (board1.quantity == board2.quantity);
}
```

Figure 13.18. Comparing two structures in C.

In contrast, with an array of structures, the array name, a subscript, and a member name are needed.[8] The other factor is whether the functions in the program process mainly rows or columns of data. Row processing functions favor the structure, while column processing functions are most practical with the parallel arrays. A mixed strategy also may be used: We might define an array of menu items for use with menu_i() or menu_c(), but group all other data about each menu choice into a structure and have an array of these structures parallel to the menu array.
The output from print_inventory is

```
          spruce 2" x 4" x 12 feet long -> 27 in stock at $8.20
          pine 2" x 3" x 10 feet long -> 11 in stock at $5.45
          pine 2" x 3" x 8 feet long -> 35 in stock at $4.20
          white oak 1" x 6" x 8 feet long -> 158 in stock at $5.80
On sale:  walnut 1" x 3" x 6 feet long -> 300 in stock at $29.50
```

13.3.4 Comparing Two Structures

Unfortunately, there is one useful operation that no C compiler supports: **comparing** two **structures** in one operation[9]. For example, if we wish to know whether sale and plank have the same values in corresponding members, we cannot write this:

The printing loop stops at n, the number of actual entries in the table. It does not print empty array slots.

```
void print_inventory( LumberT stock[], int n, LumberT onsale ) {
    printf( "\n Our inventory of wood products:\n" );

    for (int k = 0; k < n; ++k) {      // Process the filled slots of array.
        if (equal_lumber( onsale, stock[k])) printf( " On sale: " );
        else printf( "          " );
        print_lumber( stock[k] );
    }   printf( "\n" );
}
```

Figure 13.19. The print_inventory function.

[8] Interestingly, a C++ class provides the best of both worlds; an array of class objects is a coherent object itself, but class functions can refer to individual members of the object simply, without using a dot or arrow. Further examination of this topic is beyond the scope of this book.

[9] This operation is not supported because some structures contain internal padding bytes, and the contents of those bytes are not predictable. The C standards committee chose to omit comparisons altogether rather than support a universally defined component-by-component comparison operator.

These declarations are in the file `"lumber.h"`. They are used by the program in Figure 13.21. The type declaration was discussed in Figure 13.9.

```
#include <stdio.h>
#include <string.h>
#include <stdbool.h>

#define MAX_STOCK 5
#define MAX_NAME 16

typedef struct LUMBER {       // Type definition for a piece of lumber
    char wood [MAX_NAME];     // Variety of wood.
    short int height;         // In inches.
    short int width;          // In inches.
    short int length;         // In feet.
    float price;              // Per board, in dollars.
    int quantity;             // Number of items in stock.
} LumberT;

LumberT read_lumber( void );
void print_lumber( LumberT b );
void sell_lumber( LumberT* board, int sold );           // Modifies the structure.
bool equal_lumber ( LumberT board1, LumberT board2 );
void print_inventory( LumberT stock[], int len, LumberT onsale );
```

Figure 13.20. Declarations for the lumber program.

```
if (sale == plank) ...// Not meaningful in C.
```

Instead, to find out whether two structured variables are equal, each member must be checked individually. A function that does such a comparison is shown in Figure 13.18.

In the `boards_equal()` function, we compare two structured variables of type `LumberT` and return the value `true` or `false`, depending on whether the structures are equal or unequal, respectively. We use a series of `&&` operations to make this comparison. We compare each pair of members using the appropriate comparison operator for that type. Comparing members may be as simple as using `==`; more complicated, as in calling `strcmp()`; or complex, as calling another function to compare two structured components. As long as each successive pair matches, we continue testing. If any pair is unequal, the run-time system will skip the rest of the comparisons and return false.

Calling the comparison function. In the `print_inventory` function, we use a loop to process an entire array of structures, comparing each item to a sale item, and printing.

13.3.5 Putting the Pieces Together

Throughout the previous sections, many declarations, statements, and functions have been discussed. Most of these have been gathered together into a single program. The declarations from the top of the program are shown in Figure 13.20; it contains the `LumberT` structure definition and the function prototypes. The main program is found in Figure 13.21. The function definitions are in Figures 13.14, 13.15, 13.16, 13.18, and 13.19. Each block of code in the program has a comment noting the section in which it was discussed; we do not duplicate any of that discussion here. A full run of the program generates the following output sequence:

```
Demo program for structure operations in C.

Our inventory of wood products:
        spruce  2" x 4" x 12 feet long ->  27 in stock at $8.20
        pine  2" x 3" x 10 feet long ->  11 in stock at $5.45
        pine  2" x 3" x 8 feet long ->  35 in stock at $4.20
        white oak  1" x 6" x 8 feet long ->  158 in stock at $5.80

white oak price is $5.80
white oak in stock: 158
Error: can't sell 200 white oak boards; have only 158.
```

This program incorporates the declarations and code fragments used to demonstrate structure declarations and operations in Sections 13.2 and 13.3. The included file `"lumber.h"` is given in Figure 13.20. Function definitions are in Figures 13.14, 13.15, 13.16, and 13.18.

```
int main( void )
{
    puts( "\n Demo program for structure operations.\n" );

    // Structured object declarations:  Figures 13.10, 13.11, and 13.17.
    LumberT onSale;      // Uninitialized
    LumberT* p;          // Uninitialized
    LumberT plank =    { "white oak", 1, 6, 8, 5.80, 158 };
    LumberT stock[5] = { { "spruce", 2, 4, 12, 8.20, 27},
                         { "pine", 2, 3, 10, 5.45, 11},
                         { "pine", 2, 3, 8, 4.20, 35 },
                         { "" }, // Remaining members will be set to 0.
                       };
    int n = 3;           // The number of items in the stock array.

    // Access parts of a structure:  Figures 13.11, 13.12.
    stock[n++] = plank;                      // Copy into the array.
    print_inventory( stock, n, onSale );     // Nothing is on sale.

    printf( " %s price is $%.2f\n", plank.wood, plank.price );
    p = &plank;                              // p points at plank.
    printf( " %s in stock:  %i\n", p->wood, p->quantity );
    sell_lumber( p, 200 );                   // A pointer argument.

    // Function calls using structs:  Section 13.3.2.
    p = &stock[n];                   // Point to a struct in the array.
    *p = read_lumber();              // Store a struct return value.
    n++;                             // Keep the item counter current.
    print_lumber( *p );              // Echo-print stock[3].

    print_inventory( stock, n, *p ); // stock[3] is on sale.
    sell_lumber( p, 200 );           // Call by address, value.

    // Accessing an array of structures:  Section 13.3.3.
    sale = stock[2];                 // Copy a structure.
    stock[3] = read_lumber();        // Input into an array slot.
    print_lumber( stock[3] );        // Print one array element.
    sell_lumber( 3, &stock[3] );     // Call by address.
    printf( "\n Second stock item: $%.2f\n\n", stock[1].price );
    print_inventory( stock, n, onSale ); />// Process array of structs.
}
```

Figure 13.21. Structure operations: the whole program in C.

```
Reading a new stock item.  Enter the 3 dimensions of a board: 1 3 6
Enter the kind of wood: walnut
Enter the price: $29.50
Enter the quantity in stock: 300
walnut  1" x 3" x 6 feet long ->  300 in stock at $29.50

Our inventory of wood products:
          spruce  2" x 4" x 12 feet long ->  27 in stock at $8.20
          pine  2" x 3" x 10 feet long ->  11 in stock at $5.45
          pine  2" x 3" x 8 feet long ->  35 in stock at $4.20
          white oak  1" x 6" x 8 feet long ->  158 in stock at $5.80
On sale:  walnut  1" x 3" x 6 feet long ->  300 in stock at $29.50

OK, sold 200 walnut
walnut  1" x 3" x 6 feet long ->  100 in stock at $29.50

Our inventory of wood products:
          spruce  2" x 4" x 12 feet long ->  27 in stock at $8.20
          pine  2" x 3" x 10 feet long ->  11 in stock at $5.45
          pine  2" x 3" x 8 feet long ->  35 in stock at $4.20
          white oak  1" x 6" x 8 feet long ->  158 in stock at $5.80
          walnut  1" x 3" x 6 feet long ->  100 in stock at $29.50
```

13.4 Structures and Classes in C++

Structures give the programmer a way to collect related variables into a single object. This is a huge leap forward in our ability to represent data. The C++ class is another leap beyond that: it allows us to group functions with the data objects that they work on. Members of a class can be data, as in a C struct, or functions that operate on the class data members. Putting 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++.

Classes also support a high level of data protection, using the keywords `const` and `private`. Normally, class data members are declared `private`. Class functions can see and change private data members, but functions outside the class cannot. When objects are passed as parameters, the `const` keyword prevents modification by the function that was called. This gives the programmer control over data objects so that all changes to them must be made by the functions in the object's class. Functions are usually `public` so that they can be seen and used by all other parts of the program.

C++ also supports `struct`, but it is not often used[10].

13.4.1 Definitions and Conventions.

- The word *class* is used for type declarations, 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 or a pile of lumber. 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.

[10] This allows a C++ compiler to compile C code without changes. A C++ `struct` actually follows the same syntax and rules as a C++ `class` except for one detail: struct members default to public visibility, while class members default to private.

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.
- Lumber: the type of wood, dimensions, price per board, and the quantity in stock.

13.4.2 Function Members of a Class

A class supplies the*context* in which all its members are viewed and all its functions are executed. We see this context in several ways:

- The full name of a function starts with the name of its class. For example, the full name of the print function in the LumberT class is: `LumberT::print()`. We could also define a function named print in all the other classes in the application. The compiler will not get them confused, because they are called using the name of an object, and the compiler chooses the right print function for that context.
- Suppose a Pet class has a `feed()` function two instances named tweety and spot. You would feed the pets like this: `tweety.feed(); spot.feed();` The compiler looks up the spot's type, sees that spot is a Pet, and selects the feed() function for Pets.
- When the spot.feed() executes, spot defines the context in which feeding happens. Spot's needs get met; tweety's needs are not relevant to feeding spot. We say that `spot` is the implied parameter to the `feed()` function.
- Some functions are static and are actually called using the class name. For example, suppose a Game class supplies a function that gives instructions for new players. Then it would be called thus: `Game::instructions()`

The functions in a class define the way that class can be used. 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 output function is responsible for formatting and printing the data for one object to a file or to the screen. An input function is responsible for reading the data for one item from a file or from the keyboard. Every class should provide an output function that outputs all of its data members. These are called `pring()` and are almost essential for debugging.
- Calculation functions. These do useful work. They are highly varied and depend on the application. They might compute a formula, do text processing, or sort an array.
- 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[11].

Many classes also need a *default constructor* that fully initializes the object to default values, often 0's. A default constructor is necessary to make an array of class objects. 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.

[11] More advanced programmers sometimes provide copy constructors and move constructors.

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.

Accessors. There are two kinds of accessors: getters and setters. A getter function gives a caller read-only access 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 rarely appropriate to provide a getter for every data member. Doing so is evidence of poor OO design: most things that are done with data members should be done by functions in the same class.

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 and doing so is just sloppy programming.

13.4.3 Encapsulation

One of the most important properties of a C++ class is that it can *encapsulate* data. Each class has a responsibility to ensure that the data stored in its objects is internally consistent and meaningful at all times:

- Declare all data members to be private so that only functions from the same class can modify the data members.
- Use the constructor to get the data into a new object.
- Make sure that every function in the class maintains consistency. The value stored in each data member must make sense in the context of the other data members.

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[12].

Design principle 1: A class protects its members.

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.

Design principle 2: A class should be the expert on its own members.

This is why a well designed OO program does not need getters and setters: almost all the work is done by a series of smaller and smaller classes.

Delegation In an OO program, much or most of the work is done by *delegation*. The main program creates an object, A then calls functions that belong to A's class to get the work done. A often has smaller objects (B and C) within it, so A's functions call functions in the classes of B and C to get the work done. Work orders get passed down the line from `main()` to all the other functions.

Design principle 3: Delegate the activity to the expert.

This is why a well designed OO program does not need getters and setters: almost all the work is done by a series of smaller and smaller classes.

[12] Later, you will also learn about protected variables.

13.4.4 Instantiation and Memory Management

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`. We deallocate it thus: `delete formerStu;`

> Design principle 4: Creation and deletion. With dynamic allocation, new objects should be created by the class that will contain them, and that class is responsible for deleting them when they are no longer needed.

13.5 The Lumber Program in C++

This is the C++ analog of the C program in the previous section. The output is the same, and the code is parallel throughout. However, the C++ version is written to follow the design principles of encapsulation, expertise, delegation, and creation. The discussion below will focus on the differences between the C and C++ versions.

Compile modules and files. This program is organized into two modules: a main module (main.cpp) and a lumber class module (lumber.hpp and lumber.cpp). The compiler will translate each .cpp file separately, then link them together and also link with the standard libraries. This modular organization is used for all C and C++ programs except the very small ones.

Header files (.hpp) contain only class, enum, and typedef declarations. They do not contain variables or arrays. The functions are prototyped within the class declaration, and normally defined in a separate code file (.cpp). This type information is needed at compile time to translate function calls, so any module that calls a function in class *A* must `#include` the header file for class *A*. One `#include` statement is written at the top of each .cpp file.

The major difference between C and C++ code is the way functions are called. A C function has all of its parameters inside the parentheses that follow the function name. In contrast, a C++ has an *implied parameter* whose name is written *before* the function name. The rest of the parameters are written between the parentheses.

Notes on Figure 13.22: The header file for the LumberT class.

- Note that the file name is written in the upper-right corner of each file. Please adopt this very useful habit.
- The data members are the same as the C version except that we are using a C++ string instead of a char array for the kind of wood. This makes input easier.
- The function prototypes are inside the class declaration (lines 15..24).
- This class has two constructors, one with parameters, one without (lines 15..16).
- A default constructor is prototyped and defined on line 16. The keyword `= default` makes it very convenient to initialize everything in the object to 0 bits.

```
 1  // Header file for the lumber class                           file: lumber.hpp
 2  #include "tools.hpp"
 3  #define MAX_STOCK 5
 4
 5  class LumberT {          // Type definition for a piece of lumber
 6    private:
 7      string wood;         // Variety of wood.
 8      short int height;    // In inches.
 9      short int width;     // In inches.
10      short int length;    // In feet.
11      float price;         // Per board, in dollars.
12      int quantity;        // Number of items in stock.
13
14    public:
15      LumberT( const char* wo, int ht, int wd, int ln, float pr, int qt );
16      LumberT() = default;
17      void read();
18      void print()  const ;
19      void sell( int sold );          // Modifies the object.
20      bool equal( const LumberT board2 ) const ;
21
22      string getWood()  const { return wood; }
23      float getPrice()  const { return price; }
24      int getQuantity() const { return quantity; }
25  };
```

Figure 13.22. The LumberT Class in C++

- Note that the function names here are shorter than in the C program. This is possible because the class name, LumberT, always is part of the context in which these functions are called.
- A class function is called using the name of a class instance. That class instance is the object that the function reads into or prints or compares. This will become clearer when `main()` is discussed.
- Lines 22. . . 24 are accessor functions. Although there are six data members, there are only three getters. Getters are simply not needed for many class members.
- There are no setters at all. Data is put into objects by using the constructor with parameters or using assignment to copy one entire object into another.

Notes on Figure 13.23: The lumber main program. This Figure has the main function and a print function called by main.

- Line 30 includes the header for the tools module. You will see this in most C++ programs in this text. The tools header brings in all the other header files you are likely to need. The tools module supplies three functions that are extremely helpful:
  - `banner()` prints a heading on your output, including your name and the date of execution[13].
  - `bye()` prints a termination comment that proves to your instructor that your program terminated normally.
  - `fatal()` prints a error comment, flushes the output buffers, closes your files, and aborts execution cleanly. You should call `fatal()` any time the program cannot continue to do its work. It is called with a format and an output list like `printf()`. Your format string should give clear information about the nature of the error.

[13] To make this work, you need to enter your own name once on line 2 of tools.hpp.

```
// ------------------------------------------------------------------------
// A C++ program corresponding to the Lumber Demo program in C.    main.cpp
// ------------------------------------------------------------------------
#include "tools.hpp"
#include "lumber.hpp"
void printInventory ( LumberT stock[], int n, LumberT& onSale );

// ------------------------------------------------------------------------
int main( void ) {
    cout <<"\n Demo program for structure operations in C++.\n";

    // Struct object declarations: ----------------------------------------
    LumberT onSale;     // Initialize using the default constructor.
    LumberT* p;         // Uninitialized
    LumberT plank( "white oak", 1, 6, 8, 5.80, 158 );  // Call first constr.
    LumberT stock[5] = { // Call first constructor three times.
                         { "spruce", 2, 4, 12, 8.20, 27},
                         { "pine",   2, 3, 10, 5.45, 11},
                         { "pine",   2, 3, 8, 4.20, 35 },
                         {} // Call default constructor.
                         // Call default constructor for additional elements.
                        };

    int n = 3;                    // The number of items in the stock array.
    stock[n++] = plank;                              // Copy into the array.
    printInventory( stock, n, onSale );              // Nothing is on sale.

    // Access parts of a structure ----------------------------------------
    cout <<fixed <<setprecision(2) <<" "                    // Use object part.
         <<plank.getWood() <<" is $" <<plank.getPrice() <<"\n ";
    p = &plank;                                             // Point to object.
    cout <<plank.getWood() <<" in stock: " <<p->getQuantity() <<endl;
    p->sell( 200 );                                         // Use the pointer.

    // Use a struct object to call a struct member function. --------------
    p = &stock[n];              // Point to an object.
    p->read();                  // Read into stock[4].
    p->print();                 // Echo-print stock[4].

    printInventory( stock, 5, *p );   // stock[3] is on sale.
    p->sell( 200 );                   // This will modify the sale object.

    // Test if two structures are equal; Section 13.3.4. -----------
    printInventory( stock, MAX_STOCK, onSale );
}

// -----------------------------------------------------------------
// This function is global, not part of the LumberT structure.
void printInventory ( LumberT stock[], int n, LumberT& onSale ){
    cout <<"\n Our inventory of wood products:\n";

    for (int k = 0; k < n; ++k) {     // Process every slot of array.
        if ( onSale.equal( stock[k]) )  cout <<" On sale: ";
        else cout <<"          ";
        stock[k].print();
    }
    cout <<endl;
}
```

Figure 13.23. The Lumber program in C++

- Line 31 includes the header file for the lumber module. You must do this to be able to declare lumber objects and call the lumber functions.
- Line 32 is the prototype for the only function that does not belong in the lumber class.
- Lines 39–48 create the same lumber objects as in the C program.
- Lines 50–52 are exactly the same as the Cversion.
- Compare lines 55–56 to the `print_lumber()` function in C. C++ output is quite different from C.
  - `fixed` means approximately the same thing as `%f`.
  - To print an amount in dollars and cents, Use `fixed` and `setprecision(2)`.

  Many people believe that C provides easier ways to control formatting, especially for types float and double.c
- Lines 63 calls a class function using p, a pointer to a class object. In the C++ program, the object name is first, and the function reads info directly into the object p points at. This is a different mode of operation from the C program, which reads the data into a local temporary and returns a copy of that temporary that is finally stored in the object p points at. The C++ program is more direct and more efficient.
- On lines 64 and 67, again, the pointer p provides the context in which the `print()` and `sell()` functions operate, and the functions called are the ones defined for p's object. In the C program, `p` is written as a parameter to the print function and the sell function.

Notes on the `printInventory()` function. The print function is not in the Lumber class because it involves an entire array of lumber.

> Design principle 5: The functions that belong in a class are those that deal with one class instance.
> A function that handles many class instances should be in a larger class or in main.

- The first two parameters to this function are the array of lumber and the number of valid, initialized entries in that array. Every function that processes an array must stop processing at the end of the valid entries. These parameters are passed by value, that is, copies are made in the stack frame for the function.
- The third parameter is a `LumberT` object that is passed by reference (`&`), that is, the address of the object is passed to the function and the function uses main's object indirectly. Call by reference avoids the time and space expense of making a copy and is normally used for large objects. Within the function, the syntax for using this parameter is the same as the syntax used with call by value.
- Line 79 calls the `equal()` function in the lumber class because `onSale` is declared to be a LumberT object. The `equal()` function will compare two lumber objects: (a) `onsale`, and (b) one entry in the stock array.
- On the basis of the outcome, the printInventory function will either print some spaces or print the words "On sale:", then the array entry will be printed.

Notes on Figure 13.24: The functions for the LumberT class.

- Within a class function, the name of a class member refers to that member of whatever object was used to call the function. That object is called the *implied parameter* and it forms the context in which the function exedutes.
- The constructor with parameters. Lines 85–92 define the primary class constructor. Note that a constructor does not have a return type. Each line takes the value of one parameter and stores it in the newly-created class instance. The parameter names here are short versions of the corresponding member name. That is OK.
- Another common style is to use the same name for the parameter and for the class member. In that case, the assignments would be written like this: `this->wood = wood;` "this" is a keyword. It is a pointer to the implied parameter, and `this->` can be used to select a part of the object you are initializing.

```
#include "lumber.hpp"                                      // file: lumber.cpp

// Constructor:  -------------------------------------------------------------
LumberT::LumberT( const char* wo, int ht, int wd, int ln, float pr, int qt ) {
    wood = wo;
    height = ht;
    width = wd;
    length = ln;
    price = pr;
    quantity = qt;
}

// -----------------------------------------------------------------
// Read directly into the object -- no need for a temporary variable.
void LumberT::read() {
    cout <<" Reading a new item.  Enter the 3 dimensions of a board: ";
    cin >>height >> width >>length;
    cout <<" Enter the kind of wood: ";
    cin >> ws;
    getline( cin, wood );
    cout <<" Enter the price: $";
    cin >>price;
    cout <<" Enter the quantity in stock: ";
    cin >>quantity;
}

// -----------------------------------------------------------------
void LumberT::sell( int sold ){
    if (quantity >= sold) {
        quantity -= sold;
        cout <<" OK, sold " <<sold <<" " <<wood <<endl;
        print();
    }
    else {
        cout <<" Error: can't sell " <<sold <<" boards; have only "
             <<quantity <<"\n";
    }
}

// -----------------------------------------------------------------
void LumberT::print() const {
    cout <<" " <<wood <<":  "
         <<height <<" x " << width <<", "<<length  <<" feet long -> "
         <<quantity <<" in stock at $"
         <<fixed <<setprecision(2) <<price <<endl;
}

// -----------------------------------------------------------------
bool LumberT::equal(  const LumberT board2 ) const {
    return (wood     == board2.wood )  &&
           (height   == board2.height) &&
           (width    == board2.width)  &&
           (length   == board2.length) &&
           (price    == board2.price)  &&
           (quantity == board2.quantity);
 }
```

Figure 13.24. The Lumber functions in C++

Use this diagram to understand the next program.

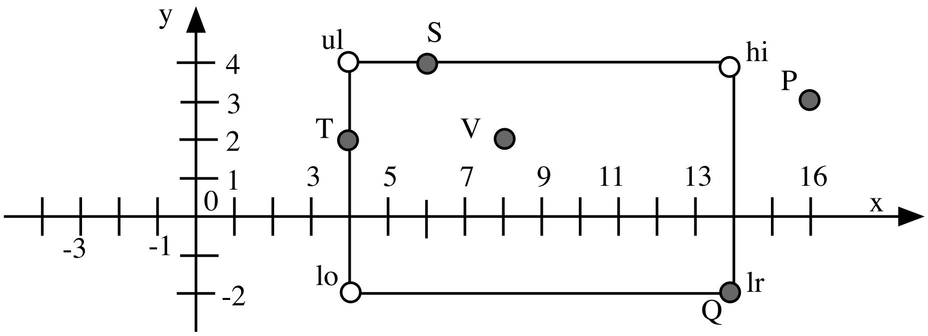


Figure 13.25. A rectangle on the *xy*-plane.

- The `read()` function. This is very much easier to read and write than the `scanf()` statements in the C code: no format codes, no ampersands, no fuss. Also, there is no local variable and no return statement. The data that is read is stored in the implied parameter and is returned that way.
- Note how easy it is to read input into a C++ string (lines 100-101). There is no need to worry about buffer overflow because strings automatically resize themselves, if needed. The `>>ws` on line 44 will eliminate the newline character on the end of the previous line plus any whitespace before the visible characters on the current line.
- The `sell()` function is parallel to the C version. Only the output statements have been changed. Notice that `print_lumber( *board )` in C is simplified to `print()` in C++. We don't need a parameter because the object used to call `sell()` replaces it. We don't need to write *anything* in front of the function name, because it is called from another function in the same class.
- The `print()` function is parallel to the C version, although the code looks quite different. In C++, class members can be used directly by any class function. So we write `height`[14] instead of using a parameter name, as in `b.height`.
- The `equal()` function is exactly like C version except for `board1.` written in front of each member name. The role played by `board1` is covered by the implied parameter in C++

13.6 Application: Points in a Rectangle

The next —pl C program example uses a structure to represent a point on the xy plane. It illustrates a variety of operations on structures, including assignment, arithmetic, and function calls.

An elementary part of any CAD (computer-aided design) program or graphics windowing package is a function that determines whether a given point in the xy plane falls within a specified rectangular region. In a mouse-driven interface, the mouse is used to select the active window, "grab" scrolling buttons, and "click on" action buttons and menu items. All these rectangular window elements on the screen are represented inside the computer by rectangles, and the mouse cursor is represented by a point. So, quite constantly, a mouse-based system must answer the question, "Is the point P inside the rectangle R?"

We present a program that answers this simple but fundamental question. The problem specifications are given in Figures 13.25 and 13.26. Figure 13.27 outlines a test plan. The various portions of the program are contained in Figures 13.30 and 13.31 through 13.34 and Figure 13.28 diagrams the structured data types used in this application. Structures are used to define both points and rectangles in this program. A point is represented by a pair of `doubles` (its x and y coordinates) and a rectangle is represented by a pair of points (its diagonally opposite corners). We use the first structured type (`point`) to help define the next. This is typical of the way in which complex **hierarchical data structures** can be constructed by combining simpler parts.

[14] We could write `this->height`, but there is no reason to write `this->`. The extra words are just clutter.

Refer to the diagram in Figure 13.25.

Problem scope: Given the coordinates of a rectangle in the integer xy plane and the coordinates of a series of points, determine how the position of each point relates to the rectangle. As an example, in the diagram in Figure 13.25, R has diagonally opposite corners at $lo = (4, -2)$ and $hi = (14, 4)$. Point $P = (16, 3)$ is outside the rectangle, $Q = (14, -2)$ is at a corner, $S = (6, 4)$, and $T = (4, 2)$ are on two of the sides, and $V = (8, 2)$ is inside the rectangle.

Limitations: The sides of the rectangle will be parallel to the x and y axes. Thus, two diagonally opposite corners are enough to completely specify the position of the rectangle.

Input: In Phase 1, the user enters the x and y coordinates (real values) of points lo and hi, the lower left and upper right corners of the rectangle. In Phase 2, the user enters the x and y coordinates of a series of points. An input point at the same position as lo will terminate execution.

Output: In Phase 1, echo the actual coordinates of the corners of the rectangle that will be used. In Phase 2, for each point entered, state whether the point is outside, at a corner of, on a side of, or inside the rectangle.

Formulas: A point is outside the rectangle if either of its coordinates is less than the corresponding coordinate of lo or greater than the corresponding coordinate of hi. It is inside the rectangle if its x coordinate lies between the x coordinates of lo and hi and the point's y coordinate lies between the y coordinates of lo and hi. The point is on a corner if both of its coordinates match the x and y coordinates of one of the four corners. It is on a side if only one of its coordinates equals the corresponding x or y coordinate of a corner and the point is not outside nor on a corner.

Computational requirements: If the user accidentally enters the wrong corners of the rectangle or enters the corners in the wrong order, the program should attempt to correct the error. The program also must function sensibly if the rectangle is degenerate; that is, if the x coordinates of lo and hi are the same, the y coordinates of lo and hi are the same, or if lo and hi are the same point. The first two cases define a line; the last defines a point. If rectangle R is a single point, then point P must be either "on a corner" or "outside" R. If the rectangle is a straight line, P can be "on a corner," "on a side," or "outside" R.

Figure 13.26. Problem specifications: Points in a rectangle.

Just as some programs take simple objects as input and operate only on those simple objects (`floats` or `ints`), conceptually, this program takes structured objects as data and operates on these structured objects. Because operations on compound objects are more complex than those on simple objects, functions are introduced to perform these operations. This keeps the main flow of logic simple and lets us focus separately on the major operations and the details of the structured objects. New function prototypes are introduced in which the `struct` and `enum` data types are used to declare both parameters and return types. Specifically, we use the enumerated type `in_type` declared in Figure 13.2, which has one code for each way that a point's position can relate to a rectangle.

A test plan for points in a rectangle. The specifications in Figure 13.26 ask us to analyze the position of a point relative to a rectangle. They require us to handle any point in the xy plane and any rectangle with sides parallel to the x and y axes. Degenerate rectangles (vertical lines, horizontal lines, and points) must be handled appropriately. A suitable test plan, then, will include at least one example of each case.

We construct such a plan, shown in Figure 13.27 by starting with the sample rectangle and points given in Figure 13.25. We then add points that lie on the other two vertical and horizontal sides. The last input for this rectangle is a point equal to the lower-left corner, which should end the processing of this rectangle. This part of the test plan is shown in the top portion of Figure 13.27. Next, we add two rectangles whose corners have been entered incorrectly, to ensure that the program will swap the coordinates properly and create a legitimate rectangle. Finally, we need to test three degenerate rectangles: a vertical line, a horizontal line, and a point.

Type of Rectangle	**Lower Left (x, y)**	**Upper Right (x, y)**	**Point (x, y)**	**Position (answer)**
Normal	(4, −2)	(14, 4)	$P = (16, 3)$	Outside
			$Q = (14, -2)$	Corner
			$S = (6, 4)$	Horizontal side
			$(5, -2)$	Horizontal side
			$T = (4, 2)$	Vertical side
			$(14, -1)$	Vertical side
			$V = (8, 2)$	Inside
			$lo = (4, -2)$	Corner, end of test
x coordinates, reversed	(2, 0)	$(-1, 5)$	(1, 1)	Inside
x coordinates, corrected	$(-1, 0)$	(2, 5)	$(-1, 0)$	Corner, end of test
y coordinates, reversed	$(-1, 5)$	(2, 0)	$(-1, 6)$	Outside
y coordinates, corrected	$(-1, 0)$	(2, 5)	$(-1, 0)$	Corner, end of test
Vertical line	(1, 0)	(1, 3)	(1, 3)	Corner
			(1, 2)	Side
			(1, 0)	Corner, end of test
Horizontal line	(0, 1)	(3, 1)	(3, 1)	Corner
			(2, 1)	Side
			(2, 0)	Outside
			(0, 1)	Corner, end of test
Point	(1, 3)	(1, 3)	(2, 0.0001)	Outside
			(1, 3)	Corner, end of test

Figure 13.27. The test plan for points in a rectangle.

13.6.1 The program: Points in a Rectangle

Figure 13.29 contains the type declarations and function prototypes for the program, and is stored in the file named `rect.h`. Corresponding function definitions are found in the file `rect.c` and in Figures 13.31 through 13.34. The `main()` program is in Figure 13.30 and the file `main.c`; it prints program titles and executes the usual work loop. The test plan is given in Figure-13.27 and diagrams of the structured types are in Figure 13.28.

Notes on Figure 13.29: Header file for points in a rectangle.

First box: type declarations.

- A header file typically begins with all the include commands that will be needed by function in this module.
- Following that are the type declarations. This placement permits all the functions prototyped below to refer to the newly defined types.

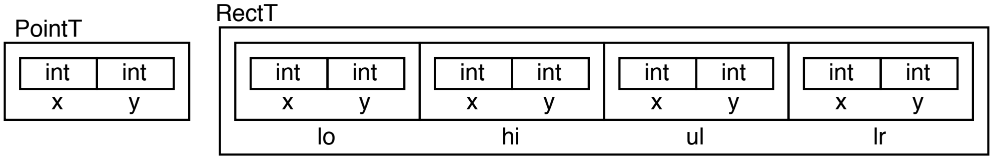


Figure 13.28. Two structured types.

This part of the program declares the necessary type definitions and function prototypes. It must be included in main.c and in rect.c.

```
#include <stdio.h>
#include <stdbool.h>
#include <ctype.h> // for tolower()
```

```
typedef struct POINT { int x, y; } PointT ;
typedef struct RECTANGLE { PointT lo, hi, ul, lr; } RectT;
typedef enum INTYPE { P_IN, P_SIDE, P_CORNER, P_OUT } InType;
```

```
bool equal( PointT p1, PointT p2 );
bool over ( PointT p1, PointT p2 );
bool under( PointT p1, PointT p2 );
bool left ( PointT p1, PointT p2 );
bool right( PointT p1, PointT p2 );
```

```
void swap( int * p1, int * p2 );
RectT makeRect( int x1, int y1, int x2, int y2 );
void testPoints( RectT r );
InType locate( RectT r, PointT p );
```

Figure 13.29. Header file for points in a rectangle.

- The type `PointT` represents the coordinates of a point in the integer xy plane. We define it as a `struct` containing a pair of integers, `x` and `y`. Although we could define this type as an array of two integers, using a structure provides a clearer and more natural way to refer to the parts. By using a *structure*, we can name the parts, not number them. This type is diagrammed on the left in Figure 13.28.
- According to the problem specifications, a rectangle is defined by the two points at its opposite corners, called `lo` and `hi`. The type `RectT`, therefore, has two `PointT` data members named `lo` (the lower-left corner) and `hi` (the upper-right corner). In addition, there are data members for the two remaining corners of the rectangle. Having these as class members is not necessary, but it makes it easy to test whether an input point is on a corner. The compound type is diagrammed on the right in Figure 13.28. We could have used an array of integers but that does not capture the fundamental nature of the problem. We prefer a structure of structures because that lets us refer to each part by a name that makes sense to a reader.
- To name the members of this structure, we choose brief but suggestive names so that the code will not be too wordy. For example, to access the x coordinate of the lower-left corner of rectangle `r`, we write `r.lo.x`.
- The third `typedef` declares `InType`, an enumeration of the positions a point can have relative to a rectangle. Each time a point is entered, the function `locate()` is called to test its position, and one of these codes is returned as the result of the function. These codes permit us to return the position information from a function with great ease and clarity. The order in which the codes are listed here does not matter.

Second box: the prototypes for PointT

- The first prototype is for the `equal()` function in Figure 13.33. It has two `PointT` parameters and compares each member of the first parameter to the the corresponding member of the other. If both pairs are equal, the points are equal, and the return value is `true` . If either pair fails to match, `false` is returned.
- The next four prototypes compare one member of `p1` to the corresponding member of `p2`. The names `over, under, left`, and `right` were chosen to make sense: `p1` can be over `p2`, under it, to its left, or to its right. These functions allow us to write brief, clear code to test the position of a point relative to a rectangle.
- The last prototype in this box is a print function. Every structure should have a print function. Sometimes it is needed for output, but it is always needed for debugging. A print function formats the data members of the class in a readable format.

Third box: the prototypes for RectT

- The first prototype is for `swap()`, defined in Figure 11.10. Swap needs no explanation at this point.

```
#include "rect.h"

// Main program for points in a rectangle ---------------------- file: main.c
int main( void ){
    char again;             // For user response in testPoints loop.
    int xLo, yLo, xHi, yHi; // To input the coordinates of the rectangle.
    puts( "\n Point in a Rectangle Program\n"
        " Given a rectangle with a side parallel to the x axis \n"
        " and a series of points on the integer xy plane,\n"
        " say where each point lies in relation to the rectangle." );

    do{
         printf( "\n Enter x and y for lower left corner:  " );
         scanf( "%i%i", &xLo, &yLo );
         printf( " Enter x and y for upper right corner:  " );
         scanf( "%i%i", &xHi, &yHi );

         RectT rect = makeRect( xLo, yLo, xHi, yHi ); // Create the rectangle.
         testPoints( rect );

         printf( "\n Do you want to test another rectangle (y/n)?  " );
         scanf( " %c", &again );
    } while (tolower( again ) != ’n’);
    return 0;
}
```

Figure 13.30. Main program for points in a rectangle.

- The second prototype is for the function `makeRect()` in Figure 13.31, which reads and validates the coordinates of a rectangle, then returns the rectangle as a structure. This function will be replaced by a constructor in the C++ version.
- The third prototype is for the `testPoints()` function in Figure 13.32. It reads in a series of points, locates those points relative to the rectangle, and prints the results.
- The fourth prototype is for `locate`, which is defined in Figure 13.34. The parameters are a rectangle and a point. The result of the function is an `InType` value; this means that the function will return one of the constants from the `InType` enumeration.
- The last prototype in this box is a print function. Every structure should have a print function for output and for debugging.

Notes on Figure 13.30: Main function for points in a rectangle.

- First box. Following that is an include command that brings in the header information (type definitions and prototypes) for the type `RectT`, used in the third box.
- This main program is in a file that is separate from the other parts of the application. The comment line identifies the purpose and the name of that file.
- Second box. A main function typically prints an identifying heading Then it creates a data object (third box) and uses that data object to do the job. If files need to be opened, main often opens them. If a query loop is needed, it is in main.
- Third box. This main program contains a query loop that processes a series of rectangles. The query loop permits this to continue as long as the user wishes.
- First inner box. Each time through the loop, the program prompts for and reads the two defining corners of a rectangle. The program prompts for the rectangle’s two corners one at a time. It would be possible to

Construct a structured object and initialize its members to the parameter values. File: `rect.c`.

```
// Read, validate, and return the coordinates of a rectangle.
RectT
makeRect( int xLo, int yLo, int xHi, int yHi ){
    RectT r;
    // If corners were not entered correctly, swap coordinates.
    if (xLo > xHi) swap( &xLo, &xHi );
    if (yLo > yHi) swap( &yLo, &yHi );
    printf( "\n Using lower left = (%i, %i), upper right = (%i, %i).\n",
            xLo, yLo, xHi, yHi );

    // Set coordinates of all four corners.
    r.lo.x = r.ul.x = xLo;                // Left side of rectangle.
    r.lo.y = r.lr.y = yLo;                // Bottom of rectangle.
    r.hi.x = r.lr.x = xHi;                // Right side of rectangle.
    r.hi.y = r.ul.y = yHi;                // Top of rectangle.

    return r;
}
```

Figure 13.31. Making a rectangle.

read both corners (all four coordinates) in one statement, but entering four inputs at one prompt is likely to cause confusion or omissions.

- Second inner box. Once the coordinates have been read in, it is possible to make a rectangle. The makeRect function stores the coordinates in the object it creates and returns. Then the newly-initialized object is sent to `testPoints()`, which inputs a series of points and locates them relative to the rectangle.
- Note that the main program is not involved with any of the details of the representation or processing of rectangles and points. All that **work is delegated** to other functions, and each of them accomplishes just one part of the overall job. This kind of modular program construction is the foundation of modern programming practice.

Notes on Figure 13.31: Making a rectangle. This function illustrates input, validation, and returning a structure. The coordinates were read in `main()` and are passed as parameters to this function.

First box: testing for valid input.

- The correct operation of the algorithm depends on having the coordinates of the lower-left corner be less than or equal to the coordinates of the upper-right corner. The user is instructed to enter the points in this order. However, trusting a user to obey instructions of this sort is never a good idea. A program that depends on some relationship between data items should test for that relationship and ensure that the data are correct.
- If the user enters the wrong corners (upper left and lower right) or enters the correct corners in the wrong order, the program still has the information needed to continue the process. That information must be stored in the correct members of the rectangle structure. So we test for this kind of an error and swap the coordinates if necessary.
- After initializing the corners of the rectangle properly, the points lo and hi are printed, using the usual mathematical notation. `(x, y)`.
- The program prints the lo and hi points after testing for errors because coordinates might have been swapped. A program always should echo its input so that the user knows what actually is being processed. This is essential information during debugging, when the programmer is trying to track down the cause of

Input and test a series of points. File: `rect.c`.

```
const char* answer[4] = {"inside", "on a side of", "at a corner of", "outside"};

void testPoints( RectT r ) {
    PointT pt;                  // The points we will test.
    InType where;               // Position of point relative to rectangle.
    puts( " Please enter a series of points when prompted." );
    do {
        printf( "\n Enter x and y (%i %i to quit):  ", r.lo.x, r.lo.y );
        scanf( "%i%i", &pt.x, &pt.y );
        where = locate( r, pt );
        printf( " The point (%i, %i) is %s the rectangle.\n",
                pt.x, pt.y, answer[where] );
    } while (!equal( pt, r.lo ));
}
```

Figure 13.32. Testing points.

an observed error. Being certain about the data used reduces the number of factors that must be considered. Echoing the input also gives the user confidence that the program is progressing correctly.

- As an example, if two points were entered incorrectly, the user might see the following dialog. Note that the y coordinates get corrected:

```
Enter x and y for lower left corner: -1 5
Enter x and y for upper right corner: 2 0

Using lower left = (-1, 0), upper right = (2, 5).
```

Second box: setting the rectangle's coordinates.

- Each of the four coordinates is shared by two corners of the rectangle. This is made clear by assigning each value to two points.
- The assignment statements show why we choose short names for the parts of the structures, instead of longer names like `upperRight` and `xCoordinate`. Since we may have to write a long list of qualifying member names to specify an individual component, even short names can result in long phrases.

Design principle #6: Keep it short and simple.

If member names are long, the statements are difficult to write on one line and they become hard to read and error prone. It is much easier to read a brief expression like the first version that follows than a long-winded version like the second:

```
r.lo.x = r.ul.x = xLo;
rectangle.lowerLeft.xCoordinate = rectangle.upperLeft.xCoordinate = xLo;
```

Third box: returning the rectangle. A `RectT` variable.`r`, is declared at the top of the `makeRect()` function. It is a local variable in this function and will be deallocated when the function returns. Within the function, `r`'s members are initialized to the parameter values. When control reaches the `return` statement, `r` contains eight integers representing the four corners of a rectangle.

The `return` statement sends a copy of this structured value containing these four points back to `main`, then immediately deallocates the original. This structured value must be stored in a `RectT` variable in `main`.

Notes on Figure 13.32. Testing points. We illustrate some basic techniques for using enumerations and structures: variable declarations, output, and function calls that use and return structured data.

Here are five one-line functions to compare two points.

```
bool equal( PointT p1, PointT p2 ){ return p1.x == p2.x && p1.y == p2.y; }
bool over ( PointT p1, PointT p2 ){ return p1.x > p2.x; }
bool under( PointT p1, PointT p2 ){ return p1.x < p2.x; }
bool left ( PointT p1, PointT p2 ){ return p1.y < p2.y; }
bool right( PointT p1, PointT p2 ){ return p1.y > p2.y; }
```

Figure 13.33. Comparing two points.

First box: An array of strings. To allow convenient and readable output of the results, we declare an array of strings parallel to the enumeration constants. Each string in this array is written in plain English and describes the location of a point w.r.t. the given rectangle. This array is used in the inner box.

Second box: processing points.

- We use a `do...while` sentinel loop to process a series of points. On entering the loop, the program prompts for and reads a point, explaining clearly how to end the loop (by entering the point `lo` as a sentinel). When reading data into a structured variable (or printing from it), it is necessary to read or print each member of the structure separately. The program cannot scan data directly into `pt` as a whole but must scan into `pt.x` and `pt.y`. Similarly, it cannot print the lower-left corner of `r` by printing `r.lo`; it must print the individual members, `r.lo.x` and `r.lo.y`.
- In the inner box, the program calls the function `locate()` to determine the position of point `pt` relative to rectangle `r`. The two arguments are structured values. A structured argument is passed to a function in the same manner as an argument of a simple type, that is, by copying all of the structure's members. This function returns an `InType` value, which is stored in `where`, an `InType` variable.
- Since enumerated types are represented by small integers, the program could print the `InType` value directly, in an integer format. However, this would not be very informative to the user. Instead, we use the integer as a subscript to select the proper phrase from the parallel array of answers defined in the first box. The `printf()` statement prints the resulting string using a `%s` format.
- At the end of the loop, the program calls the `equal()` function to determine whether the point `pt` is the **sentinel value**; that is, the same point as the lower-left corner of the rectangle. If so, we leave the loop. The `equal()` function returns a `true` or `false` answer, which can be tested directly in a `while` or `if` statement.

  We simplify and clarify the logic of `testPoints()` considerably by putting the testing details into the `equal()` function.

Notes on Figure 13.33: Comparing two points These five functions express positions on the plane in intuitive language that relates directly to positions on the plane. Creating a function for something that can be written in one line may seem unusual. However, factoring out this detail makes the `locate()` function in Figure 13.34 much cleaner and easier to understand. This, in turn, leads to arriving at correct code much faster.

Notes on Figure 13.34. Point location. The brief but logically complex function in Figure 13.32 is called from the `testPoints()` function in Figure 13.32.

The function header.

- The two parameters are `struct` types, the return value is an enumerated type.
- We use short names for the parameters because the task of this function requires using logical expressions with several clauses. Short names let us write the entire expression on one line in most cases.

Testing for containment.

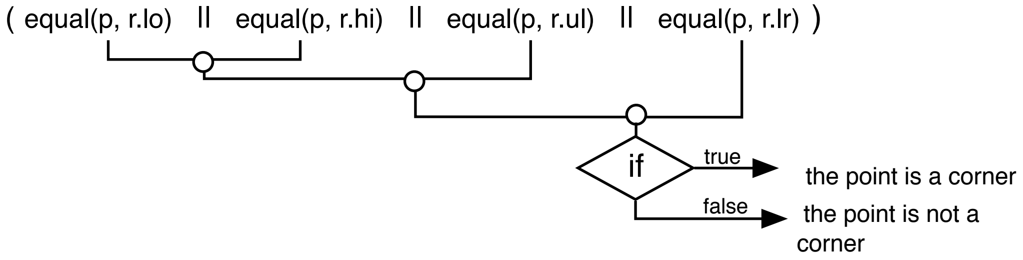
- The logic of this function could be organized in many different ways. We choose to ask a series of questions, where each question completely characterizes one of the locations.
- If a case tests `true`, the program stores the corresponding position code in an `InType` variable named `place`. Otherwise, it continues the testing.

Given a rectangle and a point, return a code for the position of the point relative to the rectangle. This function is called from the `testPoints()` function in Figure 13.32.

```
InType                                        // Where is p with respect to r?
locate( RectT r, PointT p ) {
    InType place;                             // Set to out, in, corner, or side.
    if (over(p, r.hi) || under(p, r.lo) || left(p, r.lo) || right(p, r.hi))
        place = P_OUT;
    else if (over(p, r.lo) && under(p, r.hi) && right(p, r.lo) && left(p, r.hi))
        place = P_IN;
    else if (equal(p, r.lo) || equal(p, r.hi) || equal(p, r.ul) || equal(p, r.lr))
        place = P_CORNER;
    else place = P_SIDE;
    return place;
}
```

Figure 13.34. Point location.

- Each test is a logical expression with four clauses that compares the two coordinates of `p` to the four coordinates of `r`.
- The three tests use one logical operator repeatedly. They will be parsed and executed in a strictly left-to-right order using lazy evaluation:


(equal(p, r.lo) || equal(p, r.hi) || equal(p, r.ul) || equal(p, r.lr))
if
true
the point is a corner
false
the point is not a corner

- The most complicated case to identify occurs when a point lies on one of the sides of the rectangle. A logical expression to identify this case would be even longer than the three other expressions. However, by leaving this case until last, we can identify it by the process of elimination. The final `else` clause handles a point on a side.

Program testing. The test plan presented in Figure 13.27 tests all possible kinds of rectangles (ordinary and degenerate) with points in all possible relationships to each rectangle. We show some, but not all, the test results here.

With an ordinary rectangle, the output will appear as follows. The first six lines are from `main()` and the the point prompt and processing are from `testPoints()`. The final question is again from `main()`.

```
Given a rectangle with a side parallel to the x axis
and a series of points on the xy plane,
say where each point lies in relation to the rectangle.

Enter x and y for lower left corner: 4 -2
Enter x and y for upper right corner: 14 4

Using lower left = (4, -2), upper right = (14, 4).
Please enter a series of points when prompted.

Enter x and y (4 -2 to quit): 16 3
The point (16, 3) is outside the rectangle.
```

```
Enter x and y (4 -2 to quit): 14 -2
The point (14, -2) is at a corner of the rectangle.

Enter x and y (4 -2 to quit): 6 4
The point (6, 4) is on a side of the rectangle.

Enter x and y (4 -2 to quit): 5 -2
The point (5, -2) is on a side of the rectangle.

Enter x and y (4 -2 to quit): 4 2
The point (4, 2) is on a side of the rectangle.

Enter x and y (4 -2 to quit): 14 -1
The point (14, -1) is on a side of the rectangle.

Enter x and y (4 -2 to quit): 8 2
The point (8, 2) is inside the rectangle.

Enter x and y (4 -2 to quit): 4 -2
The point (4, -2) is at a corner of the rectangle.

Do you want to test another rectangle (y/n)? y
```

A specified requirement is that the program perform sensibly with a “rectangle” that is a straight line. We tested such a degenerate rectangle; an excerpt from the output is

```
Enter x and y for lower left corner:  1 0
Enter x and y for upper right corner:  1 3

Using lower left = (1, 0), upper right = (1, 3).
Please enter a series of points when prompted.

Enter x and y (1 0 to quit):  1 3
The point (1, 3) is at a corner of the rectangle.

Enter x and y (1 0 to quit):  2 1
The point (2, 1) is outside the rectangle.

Enter x and y (1 0 to quit):  1 2
The point (1, 2) is on a side of the rectangle.

Enter x and y (1 0 to quit):  1 0
The point (1, 0) is at a corner of the rectangle.
```

A degenerate rectangle that is just a single point also gives sensible output:

```
Enter x and y for lower left corner:  1 3
Enter x and y for upper right corner:  1 3

Using lower left = (1, 3), upper right = (1, 3).
Please enter a series of points when prompted.

Enter x and y (1 3 to quit):  2 0
The point (2, 0) is outside the rectangle.

Enter x and y (1 3 to quit):  1 3
The point (1, 3) is at a corner of the rectangle.
```

13.7 Points in a Rectangle written in C++

This C++ implementation is like the C implementation in many ways:

- The basic organization is the same, with three files: the main program and a header file and code file for the two classes.
- Every code file (.c or .cpp) needs to include its matching header file (.h or .hpp). The declarations in the header file are separated from the code because the header file must also be included by any client modules The header information becomes *common knowledge* that is needed to allow two modules to be linked together and work properly.

```
//  Figure 13.35: C++ version of points in a rectangle.                main.cpp
// -----------------------------------------------------------------------------
#include <cctype>
#include "tools.hpp"
#include "rect.hpp"

// -----------------------------------------------------------------------------
int main( void ) {
    char again;         // For user response in test_points loop.
    int xLo, yLo, xHi, yHi;  // Input for the coordinates of a rectangle.

    banner();
    cout <<" Given a rectangle with a side parallel to the x axis \n"
           " and a series of points on the integer xy plane,\n"
           " say where each point lies in relation to the rectangle.\n";

    do {
        cout <<"\n Enter x and y for lower left corner: ";
        cin >>xLo >>yLo ;
        cout <<" Enter x and y for upper right corner: ";
        cin >>xHi >>yHi ;

        RectT rect( xLo, yLo, xHi, yHi );             // Create the rectangle.
        rect.testPoints();              // Create and test a series of points.

        cout <<"\n Do you want to test another rectangle (y/n)? ";
        cin >> again;
    } while (tolower( again ) != ’n’);
    return 0;
}
```

Figure 13.35. Main function in C++ for points and rectangles.

- The program logic is the same, including the treatment of mixed up and degenerate input cases.
- The C++ and C types create identical data objects.
- The output from the program is the same.

There are also several major ways that this C++ implementation differs from the C implementation. Discussion will focus on the differences.

- Data members are private.
- C++ classes have function members as well as data members. A function is called using the name of a class object.
- Inline functions are introduced. An inline function is completely defined in the header file. When this function is called, the compiler will replace the call by the body of the function. It will not do a jump-to-subroutine and return. This improves performance at run time. This is only recommended for short functions.
- The C++ classes have constructors and print functions.
- Throughout the code, `const` is used in the modern manner to limit opportunity for undetected program bugs.
- The I/O system is different.
- The syntax for `enum` declarations is simpler than C, but the usage is the same.

Notes on Figure 13.35: The main function in C++. Little comment is needed for this function because, except for the way I/O is done, it is nearly identical to the C version. It prints a title and has a query loop to allow entry of a series of rectangles. For each rectangle, the four number read as input are use to construct a rectangle, and the rectangle is used to call the `testPoints()` function.

Notes on Figure 13.36: C++ classes for points and rectangles. Compare this to the C header file in Figure 13.29.

Line 33: All of the C++ programs in this text will use functions from the `tools` module. This header file also includes all the other standard C++ library headers that we are likely to need. The functions that are prototyped here are defned in the file `rect.cpp` in Figure 13.38

Line 34: This enumeration is simpler than the C version. We don't need typedef and a typedef name at the end. The name we will use is the one that follows the keyword `enum`.

Lines 37 through 49: This is the class declaration for the point type, `PointT`. In addition to data members, it lists prototypes or definitions for all the functions that relate to points.

- Line 41 is a complete definition of a default constructor that will initialize all members of a new object to 0's. A default constructor is needed to create an uninitialized class object.
- Line 42 is the constructor with parameters that will be used to initialize points when the data is input.
- Lines 44 . . . 48 are complete definitions of inline functions. The .cpp file does not contain anything for these functions. Any function that fits on one line should be defined this way.

```
31  //  Figure 13.36: Point and Rectangle classes in C++                  rect.hpp
32  // -----------------------------------------------------------------------------
33  #include "tools.hpp"
34  enum InType{ P_IN, P_SIDE, P_CORNER, P_OUT };
35
36  // -----------------------------------------------------------------------------
37  class PointT {
38  private:
39      int x, y;                       // x and y coordinates of a point.
40  public:
41      PointT() = default;             // Default constructor.
42      PointT( int x, int y );         // Constructor with parameters.
43      void print() const ;
44      bool equal ( PointT p2 ) const { return x == p2.x && y == p2.y; }
45      bool over  ( PointT p2 ) const { return x > p2.x; }
46      bool under ( PointT p2 ) const { return x < p2.x; }
47      bool left  ( PointT p2 ) const { return y < p2.y; }
48      bool right ( PointT p2 ) const { return y > p2.y; }
49  };
50
51  // -----------------------------------------------------------------------------
52  class RectT {
53  private:
54      PointT lo;        // bottom left corner of rectangle in the x-y plane
55      PointT hi;        // top right corner of rectangle in the x-y plane
56      PointT ul;        // top left corner of rectangle in the x-y plane
57      PointT lr;        // bottom right corner of rectangle in the x-y plane
58  public:
59      RectT( int xLo, int  yLo, int xHi, int yHi );
60      void   testPoints();
61      InType locate( PointT P ) const ;
62      void print() const;
63  };
```

Figure 13.36. The C++ class declarations for points and rectangles.

```
64  // Figure 13.37: Functions for the Point class.                        point.cpp
65  // ----------------------------------------------------------------------------
66  #include "rect.hpp"
67  const char* answer[4] = {"inside", "on a side of", "at a corner of", "outside"};
68  // ---------------------------------------------------
69  PointT::PointT( int x, int y ) {
70      this->x = x;
71      this->y = y;
72  }
73  // ---------------------------------------------------
74  void PointT::print() const {
75      cout <<" (" <<x <<", " <<y <<") ";
76  }
```

Figure 13.37. C++ functions for points.

- The modifier `const` is used on after the closing) and before the opening `{` for six of these functions. A `const` in this position means that the function will not change the implied parameter. Further, if it calls other functions, *they* will not be permitted to change the implied parameter, either.
- Note that the careful layout improves readability.

Lines 22 through 33: This is the class declaration for the rectangle type, `RectT`.

- Lines 24–27 declare data members to represent the four corners of the rectangle. A comment is given on each to make its purpose clear.
- Line 29 is the constructor with parameters that will be used to initialize rectangles when the data is input.
- Lines 31 and 32 are const functions. The const keyword after the parameter list prevents any change to the implied parameter.

Notes on Figure 13.37: C++ functions for points.

Line 66: The code file point.cpp includes its the header file that contains the class declaration: rect.hpp, which is also included by the main.cpp. The header information becomes *common knowledge* that is needed to allow these two modules to work together.

Line 67: As in the `C` version, we use an array of constant strings to produce readable, English-language answers. This is an array of const char* strings, not of `C++` strings, because there is no need for the "growing power" of the `C++` string.

Lines 69. . . 72: comprise the `PointT` constructor with parameters. This kind of constructor uses the parameters to initialize the newly allocate object. In this case, there is one parameter per data member and it is very natural to give them the same names. To break the ambiguity, we refer to the data member named x as `this->x` and to the parameter as `x`. This is a common way to write a constructor.

Lines 74. . . 76: The print function in this class has no analog in the `C` version. However, a class is supposed to be the expert on its members, and that includes knowing how to print them. So we define a `print()` function that formats the data members nicely. Note that it does not put a newline on the end of the line. It is better to let the caller decide whether that newline is wanted.

Notes on Figure 13.38: C++ functions for rectangles.

Line 79: The include. As in the `C` version, every .cpp file must include its matching header file. This should be the only `#include`command in the file. All other includes should be in the header file.

```
//  Figure 13.38: Functions for the Rectangle class.                   rect.cpp
// -----------------------------------------------------------------------------
#include "rect.hpp"

// Read, validate, and store the coordinates of a rectangle.
RectT:: RectT(int xLo, int yLo, int xHi, int yHi) {
    // If corners were not entered correctly, swap coordinates.
    if (xLo > xHi)  swap( xLo, xHi );
    if (yLo > yHi)  swap( yLo, yHi );

    lo = PointT( xLo, yLo );        // These 2 corners define the rectangle.
    hi = PointT( xHi, yHi );
    ul = PointT( xHi, yLo );        // Used to test if point is on a corner.
    lr = PointT( xLo, yHi );
}

// -----------------------------------------------------------------------------
// Analyze the position of a point with respect to a rectangle in the xy plane.
//
void RectT:: testPoints() {
    InType where;      // Position of point relative to rectangle.
    PointT pt;
    int x, y;       // For inputting coordinates.

    cout <<" Please enter a series of points when prompted.";
    do {
        cout <<"\n Enter x and y ";
        lo.print();
        cout <<" to quit.  ";
        cin >>x >>y ;
        pt = PointT(x, y);
        where = locate( pt );
        pt.print();
        cout <<" is " <<answer[where] <<" the rectangle.\n";
    } while (! pt.equal( lo ));
}

// -----------------------------------------------------------------------------
// Given a point and a rectangle, return a code for the position of the point
// relative to the rectangle.
InType  RectT::
locate( PointT p ) const {
    InType place;                      // Set to out, in, corner, or side.

    if (p.over(hi) || p.under(lo) || p.left(lo) || p.right(hi)) place = P_OUT;
    else if (p.over(lo) && p.under(hi) && p.right(lo) && p.left(hi)) place = P_IN;
    else if (p.equal(lo)|| p.equal(hi) || p.equal(ul) || p.equal(lr))place = P_CORNER;
    else place = P_SIDE;
    return place;
}

// -----------------------------------------------------------------------------
// Print the coordinates of a rectangle.
void RectT::
print() const {
    cout <<"\n Lower left corner = ";
    lo.print();
    cout <<" Upper right corner = ";
    hi.print();
}
```

Figure 13.38. C++ functions for rectangles.

Lines 82. . . 91: The constructor.

- Lines 84. . . 85 check for and correct data that was entered wrong. The C++ `<algorithm>` library has an implementation of swap. There is no need to define it here in the rectangle program.
- Lines 87. . . 90 use the input values to construct points, and assign those points to the data members of the class.
- Line 104 calls a function in the point class. The implied parameter will be the point named `lo`.
- Line 108 calls a function in the current class, rectangle. We do not need to write an object name in front of the function name when a function is called from another function in its own class. The implied parameter in `locate()` will be the same object that is the implied parameter in `testPoints()`.
- This is different from the C version. In C, there is no privacy, and the `makeRect()` function simply reached into the points and set the x and the y. In C++, those parts are private and the rectangle constructor cannot access them. So it uses the input data to call the point constructor and initialize the points. This is OO: privacy matters.

Lines 96. . . 112: `testPoints()`.

- This function operates exactly like the C version. It reads a point, locates it w.r.t the rectangle, and prints the answer. Data entry and analysis is repeated until the sentinel values (the coordinates of the lower left corner) are entered.
- Lines 106 and 107 input x and y coordinates then call the point constructor to initialize a new point, which is stored in the variable `pt`. We go through the point constructor because the coordinates are private within the point and we cannot do this job the way it was done in C, by reading directly into `pt.x` and `pt.y`.
- Lines 109 and 110 print the output. In C, this can be done in one line. C++ also permits it to be done in one line by using an operator definition. (This will be covered later in the text.)

Lines 117. . . 126: `locate()`.

- Note that the return type and class name are on line 117 and the function name on line 118. This puts the function name on the left margin where is it easy to find. This style makes a lot of sense in C++, increasingly so as the programmer starts using more complex types.
- This function operates exactly like the C version. It reads a point, locates it w.r.t the rectangle, and prints the answer. Data entry and analysis is repeated until the sentinel values (the coordinates of the lower left corner) are entered.
- Lines 106 and 107 input x and y coordinates then call the point constructor to initialize a new point, which is stored in the variable `pt`. We go through the point constructor because the coordinates are private within the point and we cannot do this job the way it was done in C, by reading directly into `pt.x` and `pt.y`.
- Lines 109 and 110 print the output. In C, this can be done in one line. C++ also permits it to be done in one line by using an operator definition. This is too advanced now, but will be covered later in the text.

Lines 117. . . 126: `print()`.

- Every C++ class should have a print function. It is essential for debugging.
- This code for printing the rectangle would be a 1-line function if we provided definitions for `<<` on rectangles and points.

13.8 What You Should Remember

13.8.1 Major Concepts

- An enumerated type specification allows you to define a set of related symbolic constants.
- Use `typedef` declarations to name enumerations and struct types in C. They are not necessary in C++ for these purposes.
- Types `char` and `bool` are the most commonly used enumerated types.
- A `struct` or a `class` can be used to represent an object with several properties. Each data member of the structure represents one property and is given a name that reflects its meaning.

- A `struct` or `class` type specification does not create an object; it creates a pattern from which future objects may be created. Such patterns cannot contain initializations in C, but they can in C++.
- The basic operations on objects include member access, assignment, use as function parameters and return values, and hierarchical object construction.
- Structured types can be combined with each other and with arrays to represent arbitrarily complex hierarchical objects. A table of data can be represented as an array of structures.
- An entire structured object can be used in the same ways as a simple object, with three exceptions. Input, output, and comparison of structured objects must be accomplished by operating on the individual components.
- Design principles.

  1. A class protects its members. It keeps them private and ensures that the data stored in an object is internally consistent and meaningful at all times.
  2. A class should be the expert on its own members. Implement all the functions needed to use the class.
  3. Delegate the activity to the expert. Don't try to do things in the wrong class.
  4. Creation and deletion. With dynamic allocation, new objects should be created by the class that will contain them, and that class is responsible for deleting them when they are no longer needed.
  5. The functions that belong in a class are those that deal with one class instance.
  6. Keep it short and simple. If member names are long, code is more difficult to write, read, and debug.

13.8.2 Programming Style

Enumerations. Use enumerations wisely. Use an enumeration any time you have a modest sized set of codes to define. It might be a set of error codes, conditions, or codes relating to the data. Use `#define` for isolated constants. Often, it is wise to include one item in the enumeration to use when an unexpected error occurs. Names of the enumeration constants should reflect their meaning, as with any constant or variable. For example, an enumeration for "gender" might be `enum { male, female, unknown }`.

Truth values. Many functions and operators return truth values (`false` or `true`). Use the proper symbols for truth values; avoid using 0 and 1. Learn to test truth values directly rather than making an unnecessary comparison; for example, write `if (errorFlag)` rather than `if (errorFlag == true)`.

Structure declarations. Declare all C structures within `typedef` declarations. This simplifies their use and makes a program less error prone. Using a tag name in addition, in a `struct` specification, is an advantage with some on-line debuggers.

Long-winded names. The names of the members of a `struct` or `class` should be as brief as possible while still being clear. They must make sense in the context of a multipart name, but they need not make sense in isolation.

Privacy. In a `class` keep your data members private. Minimize the number of accessors ("getters") you define and provide a mutator ("setter") only in unusual cases.

Using `typedef` ***wisely.*** Type declarations can be overused or underused; try to avoid both pitfalls. Use of `typedef` should generally follow these guidelines:

- Use `typedef` with a `struct` type for a collection of related data items that will be used together and passed to functions as a group.
- Any type named using `typedef` should correspond to some concept with an independent meaning in the real world; for example, a point or a rectangle.
- Each `typedef` should make your code easier to understand, not more convoluted. Names of types should be concise, clear, and not too similar to names of objects.

Call by value and return by value. Some structured objects are quite large. In such cases, using call by value for a function parameter consumes a substantial amount of run time during the function call because the entire structure must be copied into the parameter. This becomes significant because functions generally are used for input, output, and comparison of structures and so are called frequently. To increase program efficiency for functions that process large structures, avoid both call by value and return by value. Replace these techniques by call by address, with a `const` modifier, where appropriate.

In this chapter, both large and small structures have been declared and used. The point structure is small and it is appropriate to use call-by-value with it. The lumber structure is much larger and should be passed by address.

Array vs. structure. Use a structure to represent an aggregate type with heterogeneous parts. For example, the `LumberT` in Figure 13.9 is defined as a structure because its five members serve different purposes and have different types. If all parts of an aggregate are of the same type, we can choose between using an array and using a structure to represent it. In this case, the programmer should ask, "Do I think of the parts as being all alike or do the various parts have different purposes and must they be processed uniquely?"

For example, assume you want to measure the temperature once each hour and keep a list of temperatures for the day. This is best modeled by an array, because the temperatures have a uniform meaning and will be processed more-or-less in the same manner, likely as a group.

Sometimes either technique can be used. For example, the rectangle defined in Figure 13.30 could have been defined as an array of eight integers. However, each `int` in the `RectT` structure has a different meaning, and the numbers in the first pair must be less than the numbers in the second pair. Giving unique symbolic names to the lower and upper corners, as well as to the x and y coordinates of each corner, makes the program easier to write and understand.

Parallel arrays vs. an array of structures. These two ways to organize a list of data objects are syntactically quite different but similar in purpose. Most applications based on one could use the other instead. An array of structures provides a better model for the data in a table. Parallel arrays are well suited for menu processing and masking; they should be used when the content and purpose the items in one array is only loosely related to the content and purpose of the other, or when one of the arrays must be used independently of the other.

13.8.3 Sticky Points and Common Errors

Enumeration constants are not strings. The words that you write in an `enum` definition are literal symbols, not strings. Use them directly; do not enclose them in quotes. Because of this, you cannot input or output the symbols directly; you must use the underlying integer values for these purposes.

Call by value. If a parameter is not a pointer (and not an array), changes made to that parameter are not passed back to the caller. Be sure to use call by address if any part of a structured parameter is modified by the function.

The dot and arrow operators. Two `C` operators are used to access members of structures: the dot and the arrow. If `obj` is the name of a structured variable, then we access its parts using the dot. If `p` is a pointer to a structured variable, we use the arrow.

Even so, programmers sometimes become confused because the same object is accessed sometimes with one operator and sometimes with the other. For example, suppose a program passes a structured object to a function using call by address. Within the main program, it is a structure and its members are accessed using dot notation. But the function parameter is a pointer to a structure, so its members are accessed within the function using an arrow. Another function might use a call-by-value parameter and would access the object using a dot.

For these reasons, it is necessary to think carefully about each function parameter and decide whether to use call by value or address. This determines the operator to use (dot or arrow) in the function's body. If you use the wrong one, the compiler will issue an error comment about a type mismatch between the object name and the operator.

Comparison and copying with pointers. Remember that a pointer to a structure is a two-part object like a string. If one pointer is assigned or compared to another, only the pointer part is affected, not the underlying structure. To do a structure assignment through a pointer, you must dereference the pointer. To do a structure comparison, you must write a comparison function and call it.

13.8.4 New and Revisited Vocabulary

These are the most important terms and concepts presented in this chapter:

enumerated type
enumeration constant
type declaration
type `bool`
`true` and `false`
error codes
aggregate type
structure
class
object

member
member name
member declaration
structured initializer
tag name
`typedef` name
selection expression
dot operator (member access)
arrow operator (member access)
call by value or address

pointer to a structure
operations on structures
comparing structures
hierarchical structures
array of structures
parallel arrays
delegating work
expert principle
privacy principle
sentinel value

The following keywords, `C` library functions, and constants were discussed or used in this chapter:

`struct`
`class`
`enum`

`typedef`
`private`
`public`

`*` (dereference)
`->` (dereference and member access)
`.` (member access)

13.8.5 Where to Find More Information

- The semantics and usage of type `bool` in C++ are covered in the `C++: The Complete Reference` by *Herbert Schildt*. Refer to the index for page references in the current edition.
- The semantics and usage of type `boolean` in `Java` are covered in Chapter 4.2.5. of the *Java Language Specification* which can be downloaded from

  `http:// java.sun.com/docs/books/jls/second_edition/html/j.title.doc.html`

- Self-referential structure definitions create types in which one structure member is a pointer to an object of the `struct` type currently being defined. These types are presented in Chapter **??**, with extensive examples of their use in building linked lists.
- Chapter **??**, Figures **??**, **??**, and **??** illustrate recursive type definitions in which the tag name is required.
- The `sizeof` operator is also discussed in Figures 4.11, 10.5, and 12.16. It is used for memory management in Chapter 16.

Chapter 14

Streams and Files

We have been using the input functions (`scanf(), cin >>`) and output functions (`puts(), printf()` and `cout <<` without explaining much about the streams they read from and write to. In this chapter, we introduce the concept of streams: the predefined streams and programmer-defined streams. We also describe what a stream is and how to interact successfully with streams. We present stream management techniques, discuss file input and output, consider I/O errors, and present examples of a crash-resistant code.

Even with no compile-time errors and all the calculations correct, a program's output can be wrong because of run-time errors during the input or output stages. Such errors can be caused by format errors, incorrect data entry, or human carelessness. Hardware devices also occasionally fail. Three ways to detect certain input errors have been used since the beginning of this text:

1. Echo the input data to see what the program actually is processing for its computations. The values displayed may not be the values the user intended to enter.
2. Use `if` or `while` statements to test whether the input values are within acceptable ranges.
3. Write a test plan that lists important categories of input with the corresponding expected output. Use these examples of input to test the program and inspect the output to ensure correctness.

We describe one more important method of detecting input errors and show a variety of ways to recover from these mistakes.

The conceptual material is the same for C and C++, but of course, the function names and syntax are different. In this chapter, we present the C++ I/O system. The same material, explained and written in C is in Appendix **??**.

14.1 Streams and Buffers

To understand the complexities of I/O, it is necessary to know about streams and buffers. In this section we describe these basic components of the I/O system.

14.1.1 Stream I/O

The input or output of a program is a sequence of data bytes that comes from or goes to a file, the keyboard, the video screen, a network socket,[1] or, possibly, other devices. A stream is the conduit that connects a program to a device and carries the data into or out of the program.

You can think of a stream as being like a garden hose. To use a hose, we must find an available one or purchase a new one. Before water can flow, the hose must be attached to a source of water on one end and the other end must be aimed at a destination for the water. Then, when the valve is opened, water flows from source to destination (see Figure 14.1). When we open a stream for reading, it carries information from the source to our program. A stream opened in write or append mode carries data from the program to a

[1] A network socket is used to connect an input or output stream to a network.

An output stream connects a program to a file or an output device. Calling an output function is like opening the valve: It lets the data flow through the stream to their destination.

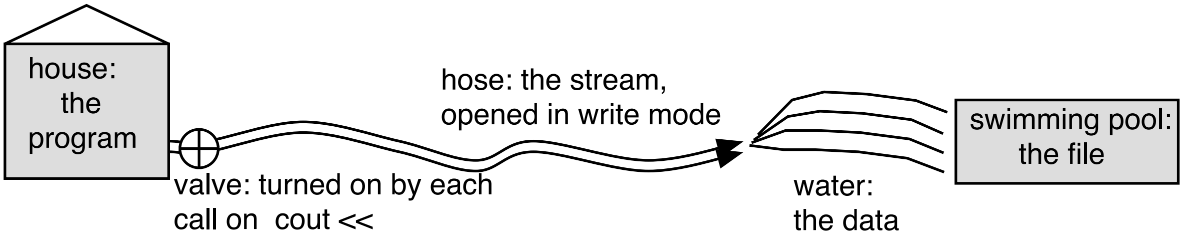

house: the program
hose: the stream, opened in write mode
swimming pool: the file
valve: turned on by each call on cout <<
water: the data

Figure 14.1. A stream carries data.

destination. Calling an input function to read the data or an output function to write them, is like turning on the valve and letting the water flow.

Why we need streams. Streams were invented to serve several important ends:

- A stream provides a standard interface for using diverse kinds of equipment. Disk files, sockets, keyboards, and video screens have very different properties, different needs, and different machine language. All of these differences are hidden by the stream interface, allowing the programmer to use one set of functions to read or write from any of these devices.
- Streams provide buffering. An input stream buffer stores blocks of input characters and delivers them to the program one line or one character at a time. An output stream buffer stores the characters that come from `puts(), printf()` and `cout <<` until enough have been collected to write a whole line on the screen or a whole block on the disk.
- Error flags. A stream is a data structure used to keeps track of the current state of the I/O system. It knows the location of the source or target device or file. It knows how much has been read from or written to the device into the buffer. It knows how much data has been read from or stored into the buffer. Finally, it contains status flags that can be tested to detect errors.

Standard streams in C++. Four streams are predefined in C++, as listed in Figure 14.2. The standard streams are opened automatically when you program begins execution and are closed automatically when it terminates. `cin` is connected, by default, to the keyboard; `cout` and `cerr` are connected by default to the monitor screen, as shown in Figure 14.3. These default connections provide excellent support for interactive input and output.

- The standard C++ input stream is `cin`. Normally, `cin` is connected to the user's keyboard, although most systems permit it to be redirected so that the data come from a file instead.
- The standard C++ output stream is `cout`. Normally, `cin` is connected to the user's video screen, although most systems permit it to be redirected so that data go to a file instead.
- The standard error stream, `cerr`, is used by the runtime system to display error comments, although any program can write output, instructions, and error messages into `cerr`. The error stream normally is connected to the user's video screen. Although it can be redirected, it seldom is.

Purpose	Stream name
	C++
Input from the keyboard	`cin`
Output to the screen	`cout`
Error output on screen	`cerr`
Logging to a file	`clog`

Figure 14.2. Standard streams.

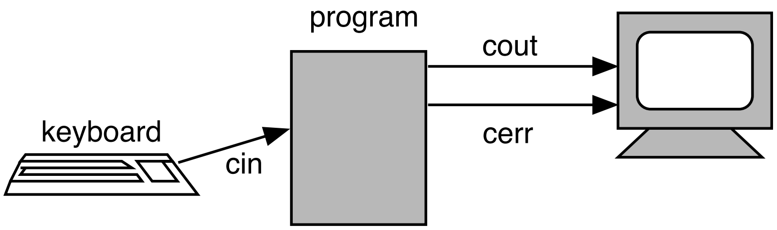


Figure 14.3. The three default streams.

14.1.2 Buffering

Water moves through a garden hose in a continuous stream. In contrast, the C++ standard input and output streams are buffered, causing the data to flow through the stream in blocks. A **buffer** is a sequence of memory locations that is part of the stream. It lies between the producer and the consumer of the data, temporarily storing data after they are produced and before they are consumed, so that the characteristics of the I/O devices can be matched to the operation of the program.

The system reads or writes the data in blocks, through the stream, even if the user inputs or outputs data one item at a time. This buffering normally is transparent to the programmer. However, there are some curious and, perhaps, tricky aspects of C++ input and output that cannot be understood without knowing about buffers.

Input buffering. An input stream is illustrated in Figure 14.4. The data in a stream flow through the buffer in blocks. At any time, part of the data in the buffer already will have been read by the program and part remains waiting to be read in the future.

An input stream delivers an ordered sequence of bytes from its source, to the program, on demand. When the buffer for an input stream is empty and the user calls for more input, an entire unit of data will be transferred from the source into the buffer. The size of this unit will vary according to the data source. For example, if the stream is connected to a disk file, at least one disk cluster (often 1,024 or 2,048 characters) will be moved at a time. When that data block is used up, another will be retrieved.

Keyboard input actually goes through two buffers. The operating system provides one level of buffering. Keystrokes go into a system keyboard buffer and remain there until you press Enter. If you make an error, you can use the Backspace key to correct it. Using the Enter key causes the line to go from the keyboard buffer into the stream buffer. Then your program takes the data from the stream buffer one item at a time.

Output buffering and flushing. An output stream delivers an ordered sequence of bytes from the program to a device. Items will go into a buffer whenever an output function is called. The data then are transferred to the device as a block, whenever the output stream is flushed, which may occur for several reasons. Information written to `cout` normally stays in the buffer until a newline character is written or the program switches from producing output to requesting input. At that time, the contents of the buffer are flushed to the screen and

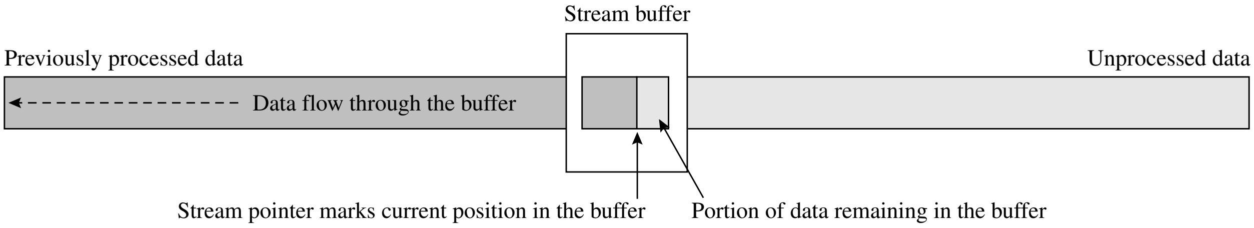

Stream buffer
Previously processed data
Unprocessed data
Data flow through the buffer
Stream pointer marks current position in the buffer
Portion of data remaining in the buffer

Figure 14.4. An input buffer is a window on the data.

become visible to the user. For other output streams, the system will send the output to its destination when the buffer is full. All output buffers are flushed automatically when the program exits.

A program also can give an explicit command to flush the buffer, of an output stream[2], which sends the data in that stream's buffer to its destination immediately. Flush is defined in the class `ostream` as a manipulator: `streamName << flush;`

In contrast to `cout`, `cerr` is an **unbuffered** output **stream**. Messages sent to it are passed on to their destination immediately. When both standard streams are directed to the same video screen, it is possible for the output to appear in a different order than that which was written. This could happen when a line sent to `cout` does not end in a newline character and is held in the buffer indefinitely, until a newline is written or the program ends. Other material sent to `cerr` during this holding period will appear on the screen first.

14.1.3 Formatted and Unformatted I/O.

Input. When data is read from the keyboard or from a text file, it comes into the stream as a series of characters. If a single character, an array of chars or a string is the end goal, the raw data is just stored in the variable. This is unformatted input.

However, if the program is using `>>` and the code to read input into a numeric variable, an additional step, number conversion, must happen before storing the result in the program's variable. First, the array of characters stored in the stream buffer must be parsed. This involves skipping leading whitespace to find the beginning of the ASCII-number. Then the conversion happens, as follows:

1. Start with *ch*, the first numeric character. Set `result=0;`
2. Subtract `'0'` (the character representing zero) from *ch*. This gives a binary number between 0 and 9.
3. Add the answer to `result`.
4. Increment your loop counter and read the next character, *ch*.
5. If *ch* is not a digit, leave the loop. If *ch* is a digit, set `result *= 10`.
6. Repeat from step 2.
7. When this loop ends, `result` is the binary integer that corresponds to the original text string.

If the input variable is a `float` or a `double`, two additional actions happen during conversion:

1. The position of the decimal point, if any, is noted, and the number of digits after the decimal point, n, is counted.
2. When this loop ends, `result` is divided by 10^n.

The way whitespace in the input is handled can be controlled by using stream manipulators:

Formatted Ouput. Number conversion also happens during output when the program calls `>>` to output a numeric variable. Stream manipulators are used to control the formatting, including:

`flush`	Flush the stream buffer.
`endl`	End the line and flush it for interactive streams.
`hex` and `dec`	Put the stream in hexadecimal or decimal mode.
`setw( n )`	Output the value in a field that is n columns wide
`right` and `left`	Display the value at right or left edge of the field.
`setfill( ch )`	Use a fill character other than spaces.
`setprecision( n )`	Show n places after the decimal point.
`fixed`	Display all float and double values in fixed point notation.
`scientific`	Display float and double values in scientific notation.

Other manipulators are define in the standard classes `ios`, `iomanip`, and `ostream`. Examples of the use of several of these manipulators will be given in the programs in this chapter.

[2] Flushing is not predefined for input streams but a definition is given in the tools library.

We show how to define streams, open them, and close them.

```
#include <iostream>      // For the standard streams.
#include <fstream>       // For file streams.
#include <iomanip>       // A set of stream management and formatting tools.
#include <sstream>       // For string-streams.
```

```
ifstream fIn( "prog4.in" );
ifstream sIn;
sIn.open( "a:\\cs110\\my.in" );
```

```
ofstream sOut( "myfolder/my.out" );
ofstream fOut( "prog4.out", ios::append );
ofstream bOut( "image1.out", ios::binary );
```

```
fIn.close();
fOut.close();
```

Figure 14.5. Opening streams in C++.

14.2 Programmer-Defined Streams

A programmer who wishes to use a file must open a stream for that file. Opening a stream creates the stream buffer and connects the stream to a file.

Streams can be opened for input, output or both and they can be created in text mode or binary mode. **Text files** (in character codes such as ASCII or Unicode) are used for most purposes, including program source files and many data files. **Binary files** are used for executable code, bit images, and the compressed, archived versions of text files. In this chapter, we concentrate on using text files; binary data files are explored in Chapter 18.

14.2.1 Defining and Using Streams

Types of streams. Streams can be for input (`istream`), output (`ostream`), or both (`iostream`). The type used to declare the stream name determines what kind of stream it is. In this chapter we will use `istreams` and `ostreams`[3].

Figure 14.5 illustrate how streams can be declared and opened in C++, resulting in the configuration shown in Figure 14.6. Opening a stream creates a data structure that is managed by the system, not by the programmer. This structure includes memory for the stream's buffer, a pointer to the current position in the buffer, flags for error and end-of-file conditions (discussed later), and anything else needed to maintain and process the stream. The programmer never accesses the stream object directly, so it is not necessary to know exactly how it is implemented.

Notes on Figure 14.5: Opening streams. We declare five streams in this Figure: `fIn`, `fOut`, `sIn`, `sOut`, and `bOut`. When opened properly, these streams will allow us access to five different files. Figure 14.6 shows the configuration that would be created if only `sIn` and `sOut` were opened.

First Box: Include files. One to four header files must be included to use streams in C++.

- The `<iostream>` header gives access to the four predefined streams and to the classes used to define them: `<istream>, <ostream> and <ios>`.
- The `<fstream>` header lets us define file-streams.
- The `<iomanip>` header gives access to a set of file management tools involved with formatting, octal and hexadecimal I/O, whitespace, and flushing buffers.

[3] The use of `iostreams` is beyond the scope of this text.

This shows the three default stream assignments, plus the streams `sIn` and `sOut`, which were declared and opened in Figure 14.5.

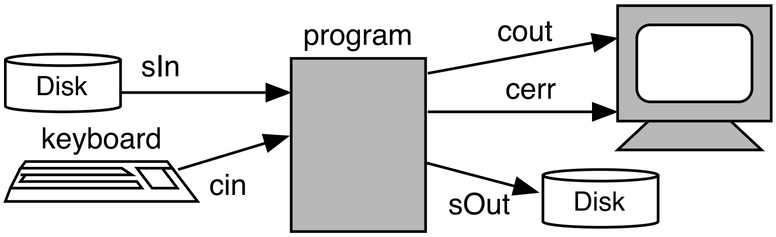


Figure 14.6. Programmer-defined streams.

- The `<sstream>` header allows us to use sophisticated input parsing and validation techniques. It is useful for applications that must process complex data. Briefly, a string is used as a temporary place to store a line of input while it is analyzed (istringstream) or createed (ostringstream), and normal stream syntax is used to get data into or out of the string. Examples will be given at the end of this chapter.

Second Box: Declaring and opening input streams. In C++, a stream can be declared and opened in the same line or separately, on two lines.

- The first line of code both declares an input stream and opens it. This is the normal way to open a file when the name of the file is known to the programmer.
- Alternatively, a stream can be declared with no initializer and opened in a later line, as in the second and third lines of code. This method is used when the name of the stream is not known prior to runtime.
- This box opens two input streams. For one, the parameter is just a file name. At run time, the file needs to be in the same directory as the executable program. This method of opening a file is strongly preferred because it is portable from one machine to another.

  For the other stream, a full pathname is given for the file. This works, but is not portable. Normally, no two file directories have the same paths in them.

Third box: Output streams. The one-line declaration with initialization, and the two-line declaration and separate call on `open()` can be used to open all kinds of streams. In this box, we show output streams with some additional opening options.

- `ofstreams` are opened in output mode. If the named file does not already exist in the named directory, it will be created. If it does exist, the old version will be destroyed. For this reason, some caution is advised; when selecting a name for an output file, it should not duplicate the name of another file in the same directory.
- On the first line, we open `sOut` and attach it to the file `my.out` in the folder named `myfolder`.
- The second line opens a stream `fOut` in and attach it to the file `prog4.out`. Material sent to `fOut` will be appended to the material that is already in `prog4.out`. If the file does not yet exist, it is created and the stream shifts to write mode. Use append mode to capture the output of several runs of your program.
- The third stream in this box, `bOut`, is opened for output in binary mode and attached to a file that will receive a binary image.
- `ios::` is the name of a base class of `iostream`. This class defines things that are common to all streams; it defines constants such as `in`, `out`, `append` and `binary` that control the mode of a stream.

Fourth box: Closing streams. All streams are automatically closed when your program terminates, and output buffers are flushed before the stream is closed. Nevertheless, it is good practice in any program to close a stream when you are done with it. Closing a stream releases the buffer's memory space and allows the file to be used by others. This becomes more and more important as the complexity and running time of a program increases.

14.2.2 File Management Errors

Stream-opening errors. When you open a stream, either in the declaration or by calling `open()`, the result is normally an initialized stream object. If the attempt to open a stream fails, the result is a null pointer.

Several things must be done to open a stream; if any one of them fails, the "opening" action will fail. These include

- An appropriate-sized buffer must be allocated for the stream. The operation fails if space for this buffer is not available in the computer's memory. If allocation is successful, the address of the buffer is stored as a member of the stream object.
- For input, the operating system must locate the file. If the file is protected, or the name is misspelled, or the path is wrong, or no file of that name exists in the designated directory, `open()` will fail.
- For output, the operating system must check whether a file of that name already exists. If so, it is deleted. Then space must be found on the disk for a new file. This process can fail if the disk is full, the file is protected, or the user lacks permission to write in the specified directory.

Because opening failure is common, it is essential to check for this condition before trying to use the newly opened stream. Otherwise, you are likely to "hang" or crash your program. Such a check is simple enough: call `is_open()` and take appropriate action if a failure is detected. This technique is shown in the next two programs. File-opening errors are handled by calling the function `fatal()`, defined in the tools library.

Stream-closing errors. Closing a stream causes the system to flush the stream buffer, do a little bookkeeping, and free the buffer's memory. It is rare that something will go wrong with this process; a full disk is the main cause for problems. Both `flush()` and `close()` return integer error indicators. However, since trouble occurs infrequently, and at the end of execution, programs rarely check for them. We will not do so in this text.

14.3 Stream Output

Once a stream has been properly opened in write or append mode, data can be sent from the program to the stream destination. The syntax for writing to any `ostream` is the same as the syntax for using `cout`. This is illustrated in the next program.

Notes on Figure 14.8: Writing a file of voltages.

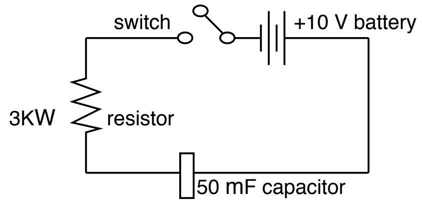
First box: The name of the input file. We don't absolutely need to `#define` the name of the input file. We could simply write it in the code in the fourth box. However, it is good design to put all file names where they can be found and changed easily. Also, this makes it easier to write good error comments, as in box 3. A `#define` command near the top of the program does this for us.

Second box: Other `#define` ***commands.*** These symbols `step` and `epsilon` are defined by expressions rather than by simple constants. An expression is permitted so long as it involves only known values, not variables. However, because of the way the preprocessor translates `#define`, an expression sometimes can lead to unintended precedence problems. To avoid this, enclose the definition in parentheses, as shown here, or use a `const` variable.

Third box: declaring and opening the output stream.

- We need to declare a stream for the output; here, we create one named `voltFile`. The name of the stream is arbitrary and need not resemble the actual name of the data file we write.
- We declare the stream `voltFile`, open the file `voltage.dat` in the current default directory. The file will be created if it does not already exist. If it does exist, the former contents will be lost.
- The value `nullptr` is returned by `open()` if any of the steps in creating a stream goes wrong. We check the return value to be sure the open command was successful and abort execution if it was not. We use the same `#define` name in both the open command and the following error report.

Problem scope: Calculate the measured voltage across the capacitor in the following circuit as time increases from $t = 0$ to $t = 1$ second, in increments of 1/15th second, assuming the switch is closed at time 0. Write the resulting data to a file.



Input: None.

Output required: The results are stored in the file `voltage.dat`. One data should be written per line for later use by a graphing program.. Each one should have two values, t and v, Four decimal places of precision are appropriate.

Constants: In this circuit, $V = 10$ volts, $R = 3{,}000$ ohms, and $C = 50 \times 10^{-6}$ farads. The circuit has a time constant τ (tau), which depends on the resistance, R, and the capacitance, C, as $\tau = R \times C = 0.15$ second.

Formula: If the switch is closed at time 0, the voltage across the capacitor is given by the following exponentially increasing equation:

$$v(t) = V \times \left(1 - e^{-\frac{t}{\tau}}\right)$$

Figure 14.7. Specifications: Creating a data table.

- The function `fatal()` in the tools library provides a succinct and convenient way to print an error message, hold the message on the screen until the operator takes action, flush all streams, close all streams, and terminate execution. It should be used whenever an error has been identified and the programmer does not know how to continue meaningful execution. A call on `fatal()` is like a call on `printf()`, with a format and a list of variables to print.

Fourth box: setting the stream format. If the file opening was successful, we come to the fourth box. We intend to output doubles, and we want the output to be in two neat columns like a table. To do this, we send two *stream manipulators* to the output stream. `endl` and `flush` are also stream manipulators.

- This is done outside the loop because it only needs to be done once.
- `fixed` gives us neat columns, where every double or float is output with the same number of decimal places.
- `setprecision(4)`, when used with `fixed`, specifies that 4 digits should be printed after the decimal point.

Fifth box: writing the table to the output file.

- When the primary output of a program goes to a file, it is important to display comments on the screen so that the user knows the program is functioning properly. In this program, This is the only output that will be displayed on the screen.
- Each pass through the loop writes one line of output into the file. Because the step size is a non-integer amount, the loop termination test incorporates a fuzz factor to make sure the final value of `t=1` is processed.
- Inside the loop, the program prints a line for each value of `t`. Since the ouput is two numbers, the code must explicitly output space between them, in this case it is a tab character.
- The first and last few lines of output from this program follow.

```
0.0000  0.0000
0.0667  3.5882
0.1333  5.8889
 ...     ...
0.9333  9.9802
1.0000  9.9873
```

Last box: cleaning up. As a matter of good style, the program closes the stream, and therefore the file, as soon as it is done writing it. If the programmer fails to close a file, the system will close it automatically when the program exits.

14.4 Stream Input

14.4.1 Input Functions

Previously, we have used `cin >>` to read keyboard input into variables. Here we introduce additional C++ functions for reading data. There are many variations on the ways that these functions can be used; only the most basic are listed here.

- `streamName >> var1 >> var2 ...;` formatted input.
  The method for reading formatted input from a programmer-defined stream is the same as reading from `cin`. Use this form for reading all primitive types of data: character, integer, float, double, pointer. Multiple values can be read on one line by chaining the `>> operators`.

  When the read operation starts, leading whitespace characters (newline, tab, space) are skipped. Then the visible characters in the stream are read, possibly converted to numbers, and stored in `var1`. The

Compute voltage as a function of time for the circuit in Figure 14.7.

```
#include "tools.hpp"
#define OUTFILE "voltage.dat"

#define V 10                 // battery voltage
#define TAU 0.15             // time constant for this circuit
#define step (1.0 / 15.0)    // time step for lines of table
#define epsilon (step / 2.0) // for floating comparison to end loop

double f( double t ) { return V * (1 - exp( -t / TAU )); }

int main( void )
{
    double t;                // time since switch closed
    double v;                // voltage at time t

    ofstream voltFile( OUTFILE );
    if (!voltFile.is_open()) fatal( "Cannot open output file:  ", OUTFILE );

    voltFile <<fixed <<setprecision(4);

    cout << "\n Writing data to file " <<OUTFILE <<"\n\n";
    for (t = 0.0; t <= (1.0 + epsilon); t += step ) {
         v = f( t );
         voltFile <<t <<" " << v <<endl;
    }

    voltFile.close();

    return 0;
}
```

Figure 14.8. Writing a file of voltages.

declared type of `var1` determines whether a char or a number is read. If the instruction has more than one `>> operator`, whitespace is skipped again, and the next visible field is read into `var2`, etc.

The input operator can also be used to read a c-string into an array. It skips leading whitespace and reads characters until the next whitespace char. Thus, it reads a single word. You cannot read more than one word at a time with `>>`. There is a serious problem with this operation: there is no way to limit the length of the read to the available memory space. If the word is longer than the array it is stored in, the excess characters will be stored anyway. They will overwrite whatever variable is next in memory. For this reason, it is unprofessional to use `>>` to read string input

- `getline( streamName, stringName );` unformatted input.
  Read a line from the file into a string. Enlarge the string, if necessary, to hold the whole line, and put a null terminator on the end. Do not skip leading whitespace. This is the easiest and safest way to read a line of text.
- `streamName.getline( arrayName, limit );` unformatted input.
  Read characters from a file into the array until `limit-1` characters have been read and stored, or until a newline is found. Put a null terminator on the end. Do not skip leading whitespace; reads whatever is there, character by character, and store it in the array. The newline character is removed from the stream and not stored in the buffer.
- `streamName.get( charVariable ) ;` unformatted input.
  Use this form for reading a single character if you *do not* want leading whitespace to be skipped.

Skipping whitespace. The bullet points above describe the default behavior of the functions. That can be changed and controlled using stream manipulators:

`hex` and `dec`	Put the stream in hexadecimal or decimal mode.
`ws`	Skip all whitespace chars in the stream; end at the next visible character.
`skipws`	Skip leading whitespace when doing formatted input.
`noskipws`	Stop skipping leading whitespace and read every character.

For example, to read a character without skipping leading whitespace, you might write: `cin >> noskipws >> charVariable;`

14.4.2 Detecting End of File

The program examples so far have relied on the user to enter data at the keyboard. This works well when the amount of data is minimal or each input depends on the results of previous inputs. It does not work well for programs that must process large masses of data or data that have been previously collected and written into a file for later analysis. For these purposes, we need file input. File input raises a new set of problems that require new solution techniques. These problems include recognizing the end of the data and handling input errors.

When you are using keyboard input, running out of data is usually not an issue. Interactive programs typically provide an explicit way for the user to indicate when to quit. Until the quit signal is given, the system will continue to prompt for input and simply wait, forever if necessary, until the user types something. However, when your data come from a file, the file *does* have a definite end, and the end of data must be recognized as the signal to quit.

Stream status flags. Every C++ stream has a set of three status flags: `bad, fail`, and eof. These are set to `false` when a stream is opened. The `eof` flag is set to `true` when end of file is recognized. The `fail` flag is set to true when there is a conflict between the contents of a stream and the type of the variable that is supposed to receive the input. `bad` is set to true when any other kind of error happens while reading.

The simple way to check for the end-of-file condition is to use the stream function `eof()` to read the status flag in the stream. The next section gives an example of a properly used end-of-file test.

The function `good()`; returns `true` for a read with no problems and `false` when eof or any kind of error happens. It does not distinguish between end of file and errors. Some people use `good()` instead of `eof()` to test for end of file. This practice conflates errors and eof, and causes a program to treat errors the same way a normal end of file is treated, which can be a bad idea.

Detecting eof for unformatted input. Remember that unformatted input just reads a stream of raw characters, while formatted input performs input conversions. A failed input conversion will set the `fail` flag in the stream. However, an unformatted input operation cannot set that flag. Thus, using `good()` to test for end of file makes sense with `getline` but not with `>>`.

The result returned after calling `getline` is an `istream&`. You can write a short program to prove this: if you use the return value to initialize a local `istream&` variable, then you can use the variable for testing the stream's status flags. However, this is not something that anyone would normally do. What *is* done is to use that return value to break out of a read loop. This works to read all the lines of a text file and end when eof happens:

```
    while (getline( mystream, mybuffer )) {...  process the data.}
```

It works because the istream class defines a type conversion from type `istream&` to type `bool`. If all the status bits are false, the bool will be `true`. If `eof` or `fail` or `bad` is true, the bool will be `false`, causing control to leave the loop. This is illustrated in the code example in Figure 14.10.

14.4.3 Reading Data from a File

Once an input stream has been properly opened, data can be read from it. The syntax for reading from any `istream` is the same as the syntax for using `cin`. This is illustrated in the next program, Figure 14.9, which incorporates two important techniques: opening a file whose name is entered from the keyboard and handling an end-of-file condition.

It often is necessary to retrieve analytical or experimental data stored in a file. These values usually are analyzed or processed in some manner. However, in this example, we just send them to the screen. For the sake of this program, we assume that the file consists of pairs of numbers in the output format generated

This program prompts the user to enter the name of a data file. This offers more flexibility than simply writing the file name as a literal constant in the program since it permits the program to be used with a variety of input sources. However, it introduces a new opportunity for error: the user could give an incorrect name.

Notes on Figure 14.9. Reading a data file.

First box: Header files.

- This text uses an accompanying library of useful commands and functions named `tools`. The tools header file, `"tools.h"` #includes all of the standard C++ header files that are used in the text, as well as the necessary context declaration: `using namespace std;`.
- The first tools function that we use is `fatal()`, which gives an easy and succinct way to handle a serious error and display essential information about the error.

Second box: Declarations.

- Since the name of the file will be entered from the keyboard, we need to declare a string variable to hold whatever file or path name the user enters.
- We also declare a `stream` named `input` for the data file.

Third box: Read in the file name.

- To avoid all possibility that the read operation might overflow memory, we use a C++ string, not a char array. The string will be lengthened to hold whatever is entered, no matter how long.
- The program prompts the user and reads the name of a file using the form of `getline` that works with strings. If the file exists in the current default directory, the user needs to type only its name. However, if it is somewhere else in the file system, the user must supply either a complete pathname or a pathname relative to the current directory.

Fourth box: Open the stream and check for errors.

- We use the `open()` function instead of opening the file in the declaration because the file name was input at run time.

Read and process a file that has two numbers on each line in point coordinate format (x, y).

```
#include "tools.hpp"     // for ios, iostream, fstream, iomanip, string, namespace

int main( void )
{
    double t, v;
    string fileName;       // File name or pathname of the file to read.
    ifstream input;

    // Create stream and open file.  ------------------------------
    cout <<"What file do you want to read?  ";
    getline( cin, fileName );                 // Read entire line.

    input.open( fileName );
    if (!input.is_open())
        fatal( "Cannot open input file named ", fileName.c_str() );

    for (;;) {
        input >>t >>v;                      // Read next input.  ------
        if (input.eof()) break;            // Test stream status.  ---

        cout <<t <<" " <<v <<endl;          // Echo input.  -----------
        // Process the input here, if processing is needed.  ----------
    }
    fclose( input );
    return 0;
}
```

Figure 14.9. Reading a data file.

- The program opens the stream `input` for reading whatever file the user named. A run-time error will occur if this file does not exist, if a pathname is given incorrectl, or if the user lacks permission to read it from the specified file or directory. All these conditions will be indicated by a `nullptr` value stored in `input`. The user will see this dialog:

```
What file do you want to read? junk
Cannot open input file named junk
```

- The fatal function cannot handle C++ strings, so we call the function `c_str()`, in the string class, to extract the C string from the C++ string.

Third box: reading and echoing the data.

- Each pass through the loop reads and echoes one line in the file. Further processing could be inserted in the indicated position.
- Line 2 of the loop reads the two data values from the file, converts them to type double, and stores them in `t` and `v`.
- Line 4 of the loop echo-prints the two values. It is always a good idea to echo the data, but it is especially so here because there is very little other output.

Inner box: End-of-file handling.

- The function `eof()` tests a flag set by the system when the program attempts to read data that are not there. This flag is *not* set when the last actual data value is read. To trigger an end-of-file condition, the program must try to read beyond the end of the file.

- That means that the eof test cannot be on the first line of the loop. So an `if ...break` *after* the read operation is used to end the loop.
- To be valid, the `eof()` test should be made after each line of data is read.
- An alternative that works is to call `if (!  input.good()) break;`
  The function `good()` returns true if all input requests have been satisfied and there is good data to process.
- Here is some sample output, run on the file `voltage.dat` written by Figure 14.8 (dashes indicate lines that have been omitted):

```
What file do you want to read? voltage.dat
   0.00          0.00
   0.07          3.59
   0.13          5.89
--------------------------
   0.80          9.95
   0.87          9.97
   0.93          9.98
   1.00          9.99
```

14.4.4 Using Unformatted I/O

The preceding two programs used formatted I/O because they were processing numbers. Sometimes a program processes only non-numeric text that can be read and processed one line at a time. The next example shows a simple way to handle error detection and end of file. This works for string input, but not for any other type of input. Specifically, it does not work for a character array, a single char, or for numbers.

Notes on Figure 14.10. Using unformatted input.

First box: Declarations.

- We need a string variable to store the name of the input file, which will be read from the keyboard.
- We need a string variable to store the name of the output file, which will be created by concatenating the string ".copy" to the end of the input file name.
- We need a string variable to store the input data, which will be read and processed one line at a time.

Second box: File names.

- Use a C++ string for input any time you are not sure how many characters that input will be. We use unformatted input here to read a file name from the keyboard into the string `inName`.
- The `+` operator here is string concatenation. A new string is created by copying the letters in `inName` followed by the letters in `".copy`. The string `inName` is not modified.

Third box: Opening the files.

- There is nothing new here. This code is the same as the previous example.
- Remember that every stream you open must be tested immediately. Although it would be possible to test both streams in one if statement, that is not a good idea. It is important to provide a high-quality error comment for the user – one that includes the word input or output and gives the name of the file that was not opened.

Fourth box: Using a while loop.

- Because of the way C++ handles the end of a file, it is normally not advisable to combine the input action and the test for eof. The reason is that the eof flag is not turned on until you attempt to read something that is not there. When that happens, the former contents of the input variables are not changed. So if the program goes on to process those values in the body of the loop, the last line will be processed twice. The cure is to use the infinite-for loop to do the repetition and an if...break to test for eof and leave the loop. This was shown in Figure 14.9.

- But there is another possibility when using unformatted input. The call on `getline()` returns a reference to `inStr`. When you use this reference in an `if` or `while` statement, it is converted to `false` if an error or eof happened during the read, and `true` if all is well. So the loop ends when eof happens, the eof bit gets set in the stream. Then when the `istream&` is converted to a bool, it is `false`.

14.4.5 Reading an Entire File at Once.

There are multiple variations of `getline()`. One useful variation allows us to read a small or very large chunk of data from a file in a single read operation, stopping at the first occurrence of a *sentinel character*. This is useful for parsing ordinary input, but can also be used to read an entire file.

- Choose any character (preferably a visible character) that does not occur in the file.
- Add it to the end of the file. This is the sentinel character.
- Use it as the third parameter in a call on `getline()`
- The entire file (excluding the sentinel) will be read and stored in the input buffer.

Notes on Figure 14.11. Reading an entire file. Everything is the same as the previous program except for the two boxed portions.

Copy a text file one line at a time. Text files normally end with a newline character. In this program, the output will end with a newline whether or not the input did.

```
#include "tools.hpp"        // Include <ios>, <iostream>, <fstream>, <string>
int main( void )
{
    string inName, outName, buffer;
    ifstream inStr;
    ofstream outStr;
    cout <<" What file do you want to copy?  ";
    getline( cin, inName );       // Read entire line.
    outName = inName + ".copy";   // Make a related output file name.
    inStr.open( inName );
    if (!inStr.is_open()) fatal( "Can't open input file: %s", inName.c_str());
    outStr.open( outName );
    if (!outStr.is_open()) fatal( "Can't open output file: %s", outName.c_str());
    cout << " Ready to copy the file.\n";
    // Read and write one line at a time.
    while (getline( inStr, buffer)) {   // Loop will exit on end of file.
        if (inStr.good())  outStr << buffer <<endl;
        else fatal( "Error reading input stream; aborting." );
    }
    cout << " Done.\n\n";
    inStr.close();
    outStr.close();
    return 0;
}
```

Figure 14.10. Copying a file.

Copy a text file in one operation.

```
#include "tools.hpp"            // Include <ios>, <iostream>, <fstream>, <string>
int main( void )
{
    string inName, outName, buffer;
    ifstream inStr;
    ofstream outStr;

    cout <<" What file do you want to copy?  ";
    getline( cin, inName );     // Read entire line.
    outName = inName + ".copy"; // Make a related output file name.

    inStr.open( inName );
    if (!inStr.is_open()) fatal( " Can't open input file: %s", inName.c_str());
    outStr.open( outName );
    if (!outStr.is_open()) fatal( " Can't open output file: %s", outName.c_str());

    cout << " Ready to copy the file.\n";
    // Read entire file and copy it.
    getline( inStr, buffer, '#');       // Read entire file, which ends in a #.

    if (inStr.good())  outStr << buffer <<endl;
    else fatal( " Error reading input stream; aborting." );

    cout << " Done.\n\n";
    inStr.close();
    outStr.close();
    return 0;
}
```

Figure 14.11. Reading an entire file.

First box: the call on `getline()`***.***

- The third parameter in this function call is the sentinel character. Everything up to the first occurrence of `#` will be read and stored in the string named `buffer`.
- The `#` will be removed from the stream but not stored in the buffer.

Second box: the error test and output.

- Always check whether your input operation worked correctly! In this case, if it does, we write the contents of `buffer` to the output file.
- If there was an error or an unexpected eof, we abort with an error comment.
- This program works properly whether or not there is a newline character on the last line.
- However, if the `#` is missing, the program aborts with an error comment.

14.5 Stream Errors

Files that are supposed to exist but seem not to, disk errors during reading, input conversion errors, and end-of-file conditions are situations that might occur. A **robust program** anticipates these things, checks for them, applies an appropriate recovery strategy, and continues processing if possible. Problems that cannot be handled routinely should be **flagged**; that is, the operator should be notified of the nature of the problem and the specific data that caused it. In this section and the next two, we look closely at **error detection** and how we may protect programs against some errors.

14.5.1 A Missing Newline

The paradigm for reading input from a file, presented in Figure 14.9, is simple and works reliably as long as the data file contains appropriate data. However, three kinds of errors in the data file will cause this program to malfunction.

A missing newline. If the newline character at the end of the last line of data is missing, the end-of-file flag will be turned on when the last item is read successfully. This is one read operation sooner than expected. The loop will test the flag before processing the last data set and the program will end immediately without processing the data. The result is shown here—the last line is missing:

```
What file do you want to read?  volts2.dat
   0.00          0.00
   0.07          3.59
   0.13          5.89
---------------------------
   0.87          9.97
   0.93          9.98
```

But this is not a programming error; it is a data file error. It *is* normal to end each line of a file with a newline character. The program is "correct" because it handles a correct data file properly.

It is possible to write the input loop so that it does the "right" thing, whether or not, the final newline is present. This kind of error handling is discussed in the next section.

14.5.2 An Illegal Input Character

Even worse behavior will happen if the format calls for a numeric input and the file contains a nonnumeric character. The character causes a conversion error, which sets the fail bit in the input stream. Control will return immediately, even if the program statement calls for more input. Values read before the conversion error will be stored correctly in their variables, but the variable being processed when the conversion error happened will not be changed and still will contain the data from the prior line of input. Finally, the stream cannot be used again until its error flags are reset.

We modified an input file named `volts2.dat` by changing the number `0.1333` on the third line by using letter instead of digits for 0 and 1. The result is subtly different and easy to miss: `O.l333`. Then we ran the program from Figure 14.9 on the modified data. The results follow:

```
What file do you want to read?  voltsbad.dat
0.00      0.00
0.07      3.59
0.07      3.59
....      ....
```

Even though the program tried to read line 4, nothing happened. The output went on and on, printing garbage, until the CTRL-C terminated it.

Error recovery. To read more input from the stream, two things must be done:

- Clear the error flag by calling `streamName.clear()`.
- Remove the offending character from the stream. This can be done by calling `char ch = get( );`, or by using the `streamName.ignore(1)` to remove one character.

  If the program is using keyboard input, it is desirable, after an error, to clear out any keystrokes that remain in the input buffer. This can be done with `streamName. ignore(255)`, which will clear out the entire line, up to the newline character, if the line is less than 255 characters long. Finally, the tools library defines an istream manipulator named `>> flush` that is very useful with keyboard input. It skips everything that is currently in the input buffer and leaves the stream empty.

We modified the read loop from the program from Figure 14.9 to add error detection and recovery code:

```
for (;;) {
    input >>t >>v;                         // Read next input. ------
    if (input.good()) cout <<t <<"      " <<v <<endl;
    else if (input.eof()) break;           // Test stream status. ---
    else {
        input.clear();
        input.ignore(1);
    }
}
```

The results of a run using the same modified file are:

```
What file do you want to read?  voltsbad.dat
0.00      0.00
0.07      3.59
333.00      5.89
0.20      7.36
0.27      8.31
....      ....
```

You can see that, in recovering from two consecutive errors, the resulting output became garbage. However, the program did run to completion and display enough information to permit a human to find and correct the error in the file.

Then we made a change at the end of the line instead of at the beginning: `0.1333     5.8SSS`. This is a different situation because, by the time the error happens, the system has read what seems to be two reasonable numbers. When the first 'S' is read, it stops the input operation and the number 5.8 is stored in `v`. The stream state is still good and the input is printed. Note, though, that it is not quite what it is supposed to be. The next time around the loop, a non-digit is read and it triggers a conversion error. We clear and ignore 1. This is repeated two more times. Finally the beginning of the 4th line arrives, with numbers, and the program continues normally. The results are:

```
What file do you want to read?  voltsbad.dat
0.00      0.00
0.07      3.59
0.13      5.80
0.20      7.36
....      ....
```

Data omission. Similar errors can occur if some portion of a multipart data set is missing. In some cases, this will be detected when the next input character is the wrong type. For instance, if a letter was read but the program expected a digit, the operation would fail. In simpler file formats, such as lists of numbers separated by whitespace, the program would not detect an error, but a misalignment in data groupings would cause the output of the program to be meaningless.

14.6 File Application: Random Selection Without Replacement

The need to randomize the order of a list of items arises in certain game programs; for example, those that shuffle an imaginary deck of cards and deal them. The same problem also arises in serious applications, such as the selection of data for an experiment where a randomly selected subset of data items must be drawn from a pool of possible items.

The automated quiz program presented here is a good example of the use of files, random selection, and an array of objects. Each question in the drill-and-practice quiz is represented as one object in an array. The program presents the questions to the user, checks the answers, and keeps score. It uses C++'s pseudo-random number generator[4] and a "shuffling" technique to randomize the order of the questions.

This application has a main program, a controller class, and a data class. The data class defines a data object, in this case, the relevant information about one chemical element. The controller class handles a collection of data objects, carries out the logic of a quiz game, and interfaces with `main()`.

[4] This was presented in Chapter 7, Figure 5.26.

Problem scope: Write a drill-and-practice program to help students memorize the symbols for the chemical elements.

Input file: The list of element names, symbols, and atomic numbers to be memorized. One element should be defined on each line of the file, with the number first, followed by the name and symbol, all delimited by blanks.

Keyboard input: When prompted with the element name, the user will type in an element's atomic number and symbol.

Output: The program will display the names of the elements in random order and wait for the user to type the corresponding number and symbol. After the last question, the number of correct responses will be printed.

Limitations: Restrict the program to 20 elements.

Figure 14.12. Problem specifications: A quiz.

Program specifications are given in Figure 14.12, the main program in Figure 14.13, and a call chart for the final solution is in Figure 14.14. The purpose of this particular quiz game is to help the user learn about the periodic table of elements. However, the data structures and user interface easily can be modified to cover any subject matter.

The complete program discussed here was developed using top-down design. The result is a highly modular design, as shown in the call chart of Figure 14.14. (Calls on `iostream` functions are omitted from the chart.)

This main program implements the specification in Figure 14.12. It uses the classes declared in Figures 14.15 and 14.16, and the functions defined in 14.17 and 14.19.

```
#include "tools.hpp"
#include "quiz.hpp"

int main( void )
{
    cout <<"\n Chemical Element Quiz\n ";
    string fileName;
    ifstream fin;

    cout <<"\n What quiz file do you want to use?  ";
    getline( cin, fileName );
    fin.open( fileName );
    fatal( " Error:  cannot open input file: %s", fileName.c_str() );
    Quiz qz( fin );
    int correct = qz.doQuiz();
    cout <<"\n You gave "<<correct <<" correct answers on "
         <<qz.getNq() <<" questions.\n ";

    fin.close();
    return 0;
}
```

Figure 14.13. An elemental quiz.

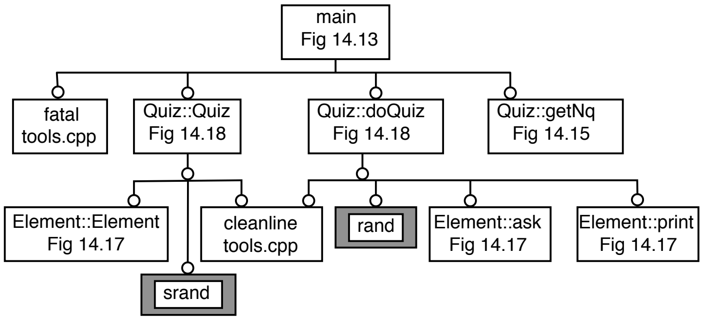


Figure 14.14. Call chart for the Element Quiz

Notes on Figure 14.13: An elemental quiz.

- The main program prints opening and closing messages. Both are important to keep the human user informed about the state of the computer system.
- It both opens and closes the data file. These two actions should both be in main or be in the constructor and destructor of the same class.

First and third boxes: local declarations.

- Most local variables should be declared at the top of a function where they can be easily found. However, sometimes it is wise or even necessary to wait until the program has the data needed to initialize the variable.
- There are three local variables in this function: a string for the filename that the user will enter, a stream to bring in the data, and a Quiz object. The stream and the string are declared in Box 1, at the top.
- The Quiz is declared later, in Box 3 because the Quiz constructor calls for a stream parameter, and the stream is opened in the Box 2.

Second box: opening the input stream.

- The data file can be opened here or in the Quiz class. We open it int main to demonstrate how to pass an open stream to a class constructor.
- Main carries out the usual opening process: (a) prompt for a file name, (b) open the input file, (c) test to be sure it is open, and (d) call `fatal()` if it cannot be opened. An error output would look like this:

```
Chemical Element Quiz

What quiz file do you want to use? q1
Error: cannot open input file: q1
```

Third box: the primary object.

- In an OO program, the job of the `main()` function is to create the first object, then transfer control to it by calling one of its functions.
- The first line here creates a Quiz object that will use the stream that was just opened.
- The second line uses the new Quiz object to call the `doQuiz()` function, which does the work.
- The return value from `doQuiz()` is used as part of the program's closing comment.

Notes on Figure 14.15. The Quiz class.

This is a controller class. Only one Quiz object will be created by `main`. It will carry out the logic of the quiz game.

First box: data members.

- A controller class usually manages an input stream and often also an output stream. Sometimes it opens both streams (and later closes them). Sometimes the streams are opened and closed in `main()`.
- A controller normally has one or more collections of data objects. In this case, an array of Elements.
- The remaining data members are used to manage the game or business logic. In this case, we have integers to store the number of questions in the quiz and the number that the user answered correctly.

This file declares the Quiz class and is included by Figures 14.13 and 14.17.

```
#include "tools.hpp"                                    // File:  quiz.hpp
#include "elem.hpp"
// ----------------------------------------------------------------------
#define MAXQ  20              // Maximum number of questions in the quiz
class Quiz {
private:
    istream& fin;
    Element elem[MAXQ];
    int nq = 0;     // Number of questions in the quiz.
    int score = 0;  // Number of correct answers.

public:
    Quiz( istream& openFile );
    int doQuiz();
    int getNq(){ return nq; }

};
```

Figure 14.15. The Quiz class declaration.

- They are initialized here in the class declaration because we know the correct initializations and because it makes writing the constructor easier to do it here.

Second box: function members.

- Prototypes and one-line functions are given in the class declaration.
- Every class has one constructor, often two. The responsibility of a constructor is to initialize the data members of the class so that the class is ready to use. This often includes reading in a file of data.
- In a controller class, the constructor is called from `main()`. When construction is finished, `main()` calls the controller's primary work function, in this case named `doQuiz()`. Control returns from `doQuiz()` to `main()` at the end of execution, and `main()` cleans up and terminates.
- Because these two functions are long, they are defined in the implementation file for Quiz, not here in the header.
- The third function here is a one-line getter. It gives read-only access to a private data member of the Quiz object.

Notes on Figure 14.16. The Element class. This header class is included by the main program in Figure 14.13 and by the Element implementation file in 14.17.

First box: data members.

- This is a data class: it has data members that represent the significant properties of a real-world object, and functions that facilitate the use of the data object.
- From the program specification, we see that three data members are needed.
- None of these members are initialized in the class declaration because all of them must be read from the input file at run time.
- We use strings (not char arrays) because we need to input the element name and symbol. Input with `string` is easy. Input with `char[]` is not easy and has been the cause of many security vulnerabilities. The current security recommendation is to use type `string` any time a variable is used for input.
- Type `short` is used instead of `int` because we know that atomic numbers will not exceed 3 digits.

This header file is included by the main program in Figure 14.13 and the class implementation in Figure 14.19.

```
#include "tools.hpp"                                    // File:  elem.hpp

class Element {
private:
    string name;              // name of element.
    string symbol;            // symbol, from periodic table.
    short number;             // atomic number.

public:
    Element() = default;
    Element( string nm, string sy, short at );

    bool ask();
    void print( ostream& out);
};
```

Figure 14.16. The Element class declaration.

Second box: constructors.

- The first constructor is a default constructor. It initializes the whole object to 0 bits. We need a default constructor here because the client class, Quiz, creates an array of Elements. To create an array of a class type you must provide a default constructor.
- The second constructor is the one that will be used to create actual Element objects. It receives three pieces of data as parameters and stores them in the corresponding three data members.

Third box: function members.

- Design principle: a class should protect its members.
  Design principle: a class should be the expert on its own members.
- The data members of Element are private (principle 1). The functions defined here, in the public area, must provide all appropriate ways for Quiz (the client class) to use Elements. A client class should *not* be reaching down into an Element object to handle its parts.
- The `ask()` function is called from `doQuiz()` each time a new random question is selected. Then `ask()` carries out the interaction with the user, prompts for answers, corrects wrong answers, and returns a true/false result when the answer was right/wrong.
- Every class needs a print function. `Element::print()` is called from `doQuiz()` each time a users answer is wrong. It formats the element data nicely and sends it to whatever output stream is specified by the parameter.

  However, even if the client did not need to print the data, the class should still have a print function. During the construction and debugging phases, it is needed.

Notes on Figure 14.17. Implementation of the Element class. This is a data class: it has data members that represent the significant properties of a real-world object, and functions that facilitate the use of the data object.

First function: the constructor. This is the simplest possible kind of constructor. It takes each parameter and stores it in the corresponding data members. Every member has a corresponding parameter.

These functions are called from the main program in Figure 14.13. They rely on the header files declared in Figures 14.15 and 14.16.

```
#include "elem.hpp"                                  File:  elem.cpp
// ----------------------------------------------------------------------------
Element::  Element( string nm, string sy, short at ){
    name = nm;
    symbol = sy;
    number = at;
}
// ----------------------------------------------------------------------------
bool Element::
ask() {
    string symb ;                      // User response for atomic symbol.
    short num;                         // User response for atomic number.
    cout <<"\ Element:  " <<name <<"\n";
    cout <<"\t symbol ?  ";
    cin >> ws;
    getline( cin, symb );
    cout <<"\t atomic number ?  ";
    cin >> num;
    return (symbol == symb && number == num );
}
// ----------------------------------------------------------------------------
void Element::
print( ostream& out ) {
    out <<name <<" is " <<symbol <<":  atomic number " <<number;
}
```

Figure 14.17. The implementation of the Element class.

Second function: `ask()`.

- This is an ordinary input function: prompt for a series of inputs, read each one, and store it in a class data member.
- One of the inputs is a string. To read it, we use string-getline. But `getline()` does not skip leading whitespace and cannot be combined with `>>` in an input chain. Therefore, getting the data requires two lines of code: `cin >> ws;` `getline( cin, symb );`
  The other input is a number and can be read simply.
- The return statement compares two strings. Because they are C++ strings, we can use `++`. If they were C-strings, we would have to use `strcmp` instead.
- The last line returns the result of the `&&` operation. No if statement is needed; don't use one.

Notes on Figure 14.18. The Quiz constructor.

First box: the function header and ctor initializer.

- The job of a constructor is to initialize the new class object so that it is fully functional and ready to use. For Quiz, this means reading an input file.
- The input file was opened in `main()` and passed as a parameter to the constructor. The type of the parameter is `istream&`. This presents a new situation and requires a new syntax.
- One of the Quiz class members is an `istream&`. One of the properties of a `&` variable is that you must use initialization, not assignment, to give it a value. A `ctor` is a syntax for initializing a class member.

These functions are called from the main program in Figure 14.13. They rely on the header files declared in Figures 14.15 and 14.16.

```
#include "quiz.hpp"                                        File:  quiz.cpp
// ---------------------------------------------------------------------
Quiz::Quiz( istream& openFile ) : fin(openFile) {
    short number;
    string name, symbol;
    for (nq = 0; nq < MAXQ; ) {
        fin >> name >> symbol >> number;
        if (fin.good()) {
            elem[nq] = Element( name, symbol, number );
            ++nq; // all is well -- count the item.
        }
        else if (fin.eof()) break;
        else {
            fin.clear();
            cerr <<" Bad data while reading slot "<< nq <<endl;
            cleanline(fin); // Skip remaining characters on the line.
        }
    }
    srand( time( NULL ) );            // initialize random number generator.
}
```

Figure 14.18. The Quiz Constructor.

- Ctors are written after the end of the parameter list in a constructor and before the opening `{` for the class.
- To write a ctor, write a colon followed by the name of a class member, followed by the initializing expression in parentheses. In this case, the initializing expression is the name of the parameter.
- In general, ctors are used in constructors anywhere that something must be initialized, not assigned. An example is a `const` data member.

Second box: the input loop.

- This function reads a series of data lines from the designated quiz input file. Each line of data is used to construct an Element, and that element. is stored in an array of Elements, which is a data member of the class.
- There is an upper limit, `MAXQ` on the number of questions in a quiz. However, any particular file could be shorter, and there might be unreadable lines in the file. For this reason, we have a class member, `nq` that stores the actual number of well-formed Elements that have been stored in the array `elem`: `nq <= MAXQ`

First inner box: the input.

- In this box, we take tree inputs from the stream `fin`, then test the stream state. If it is still `good()`, we have successfully read three fields so we use them to create an Element.
- When we store an Element in the array, we increment the subscript. This is not done in the first line of the loop because some input lines might be bad. We want to count and store only the good lines.

Second inner box: error handling.

- If the stream state is not good, it might be because of an end of file, or it might be caused by a read error or a conversion error. The two situations require different handling.
- For end of file, we simply want to leave the loop. We have created `nq` Elements and stored them in the array. All is well.

- If any kind of error happened, we must clean up the mess. This is a 3-step process:
  - First, the stream error flags must be cleared: `fin.clear();`
  - We must inform the user about the error. The error stream is used for this purpose.
  - Finally, any part of the faulty line that is still in the stream must be removed to clear the way for reading the next correct item. The `cleanline()` function in the tools library does this job, up to the next newline character.
  - The following array contents would be the result of reading data from the file `quiz2.txt`, containing six data sets, into an array with 20 slots. We use these data for the remainder of the discussion. Figures 14.20 through 14.23 show how the array contents change during the quiz.

elem	.name	.symbol	.number
[0]	Sodium	Na	11
[1]	Magnesium	Mg	12
[2]	Aluminum	Al	13
[3]	Silicon	Si	14
[4]	Phosphorus	P	15
[5]	Sulphur	S	16
[6]	Chlorine	Cl	17
[7]	Argon	A	18
[8]	??	??	?
	...	..	..
[19]	??	??	?

Last box: random numbers. A constructor is supposed to initialize everything the class needs to function. Sometimes this goes beyond initializing the data members. This is an example. Games and quizzes rely on random numbers, and the system's random number function needs to be initialized. Here we call `srand()`, which "seeds" the random number generator.

Notes on Figure 14.19: The doQuiz() function.

Outer box: Asking all the quiz questions.

- `many` is set initially to the number of Elements in the array. Each time a question is used, `many` is decremented, so it is a true count of the remaining questions. We need this number for choosing a random question.
- This large loop asks all the questions, scores the answers, keeps score, and performs housekeeping on the array of questions. Finally it returns the score to `main()`. This would not be necessary, since the score is also stored as a class member. However, it is a convenience. Many system functions use more than one path to return an answer so that the programmer can use whatever is easiest.

First inner box: Selecting a random question.

- Each time around the loop there are fewer unasked questions. The strategy here is to keep the unasked questions at the low-subscript end of the array. (See notes on fourth inner box.)
- `rand()` returns a number between 0 and MAXINT. We want a number between 0 and `many-1` (the number of remaining questions). The mod operator gives this to us.

Second inner box: Posing the question.

- Design principle: Delegate the job to the expert.
  When a question has been selected, we need to present it to the user. The Element class is the expert on all things relating to elements, including how to ask quiz questions about them. So we delegate this task to the expert by calling a function in the Element class, using the particular element we have selected: `elem[k].ask()`
- If `ask()` returns true we add 1 to the user's score and say something nice.

This function belongs in the implementation file for the Quiz class, after the Quiz constructor.

```
int Quiz::
doQuiz() {
    int many = nq;                    // Number of questions not yet asked.
    int k;                            // The random question that was selected.
    cout <<"\n This quiz has " <<nq <<" questions.\n"
         <<" When you see an element name, enter its symbol and number.\n";
    while (many > 0) {
        k = rand() % many;            // Choose subscript of next question.
        if( elem[k].ask() ){
            score++;
            cout<<"\t YES ! Score one.\n";
        }
        else {
            cout <<"\t Sorry, the answer is ";
            elem[k].print( cout );
            cout <<endl;
        }
        // Now remove the question from further consideration.
        --many;                       // Decrement the counter.
        elem[k] = elem[many];         // Move last question into vacant slot.
    }
    return score;
}
```

Figure 14.19. The doQuiz function.

Third inner box: Always be polite.

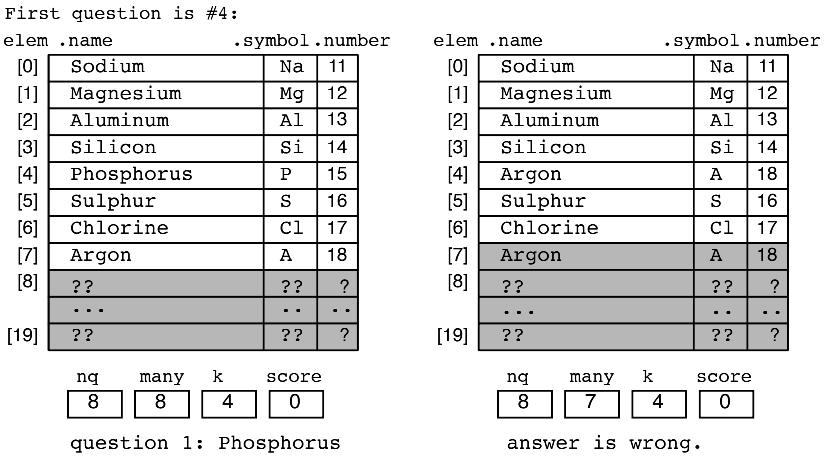
- If `ask()` returns false, we give a polite negative comment and show the correct information. After all, this IS about learning the periodic table!
- Following is the transcript of a question from one run of the program: the first three lines were written by `Element::ask()` and the last line by `Quiz::doQuiz()`, with part of the output delegated to `Element::print()`.

```
Element: Phosphorus
symbol ? Ps
atomic number ? 15
Sorry, the answer for Phosphorus is P: atomic number 15
```

- The Element class is the expert on showing information about elements. So we delegate the display job to Element by calling its `print()` function. The stream parameter could be a file-stream if we wanted file output.

Last inner box: housekeeping.

- Once a question has been used, we want to exclude it from future use. That question could be anywhere in the array, and we want to get rid of it.
- There is an easy trick for doing that. Decrement the loop counter, `many`, then take the element at subscript `many` and copy into the slot occupied by the element that was just used. We decrement first because `many` is the subscript of the first array slot that does not contain current good information.



Before asking first question.

After asking first question.

Figure 14.20. The array after one question.

- This leaves assorted garbage in the end of the array, but that is OK because we never look at anything past `many`.

Third box: remove the question from further consideration.

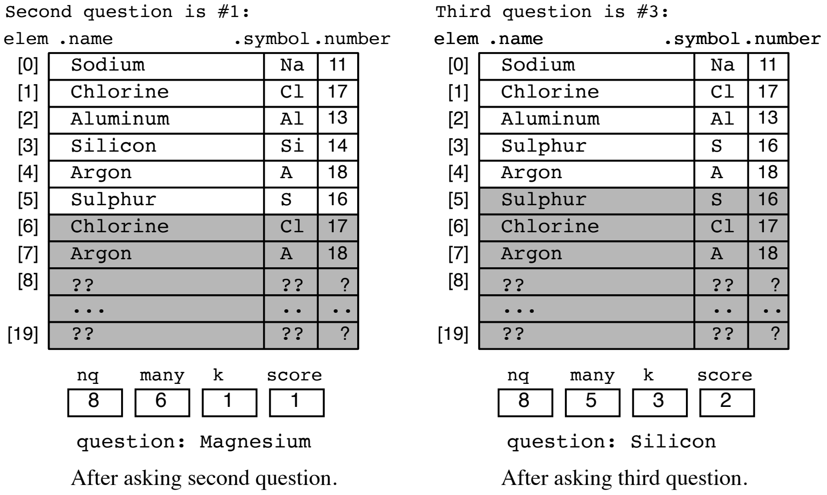
- Figures ?? through ?? show the state of the question array after asking each of the first six questions of our sample quiz. The first three were answered correctly, but only one of the next three was right.
- After each question is asked, we decrement `many`, shortening the part of the array containing the unasked questions. Just after the decrement operation, `many` is the subscript of the last unused question. We copy this question's data into slot `k`, overwriting the data of the question that was just asked.
- In the diagrams, we represent the shortening by coloring the discarded slots gray. Note in Figure ?? that, after three questions are asked, three elements no longer appear in the white part of the array and all the unused questions have been shifted up so they do appear in the white area. The contents of the gray part of the array no longer matter; they will not be used in the future because the program always selects the next question from the white part (slots `0...many-1`).
- Here is the sample dialog for questions 2 and 3, which were answered incorrectly:

```
Element: Magnesium
symbol ? Mg
atomic number ? 12
YES !  Score one.

Element: Silicon
symbol ? Si
atomic number ? 14
YES !  Score one.
```

- Note that, after the fifth question is asked in Figure 14.22, the question replacement operation effectively does nothing, since this is the last question in the list. We could have saved the effort of doing the copy operation, but a test to detect this special case would end up being more complex and more work in the long run than the unnecessary copying operation.
- The quiz continues until all questions have been presented. The last question has not been illustrated.

Fourth box: The return. When all the questions have been asked, `doQuiz()` returns the score to `main()`, which prints the results:

```
You gave 7 correct answers on 8 questions.
```



Second question is #1:

elem	.name	.symbol	.number
[0]	Sodium	Na	11
[1]	Chlorine	Cl	17
[2]	Aluminum	Al	13
[3]	Silicon	Si	14
[4]	Argon	A	18
[5]	Sulphur	S	16
[6]	Chlorine	Cl	17
[7]	Argon	A	18
[8]	??	??	?
	...	..	..
[19]	??	??	?

nq	many	k	score
8	6	1	1

question: Magnesium

After asking second question.

Third question is #3:

elem	.name	.symbol	.number
[0]	Sodium	Na	11
[1]	Chlorine	Cl	17
[2]	Aluminum	Al	13
[3]	Sulphur	S	16
[4]	Argon	A	18
[5]	Sulphur	S	16
[6]	Chlorine	Cl	17
[7]	Argon	A	18
[8]	??	??	?
	...	..	..
[19]	??	??	?

nq	many	k	score
8	5	3	2

question: Silicon

After asking third question.

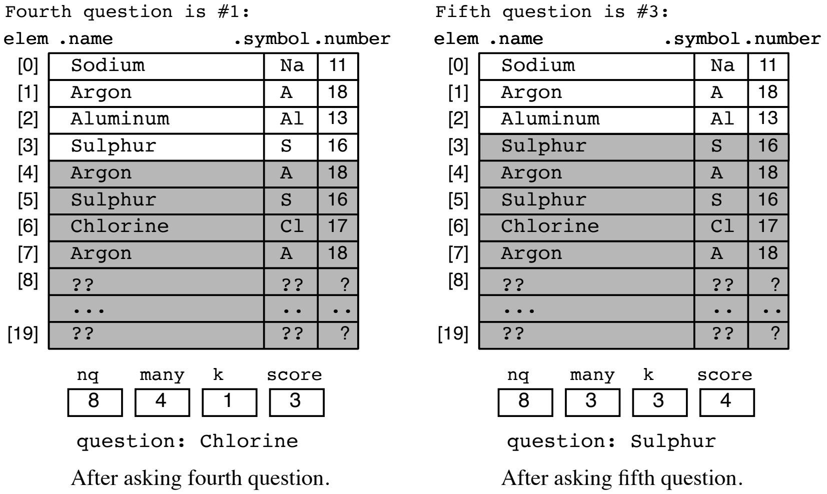
Figure 14.21. The array after three questions.

14.7 What You Should Remember

14.7.1 Major Concepts

Streams and files.

- A file is a physical collection of information stored on a device such as a disk or tape. It has a name and its own properties. A stream is a data structure created and used during program execution to connect to and access a file or any other input source or output destination.
- There are three standard predefined streams—`cin, cout, cerr`, and `clog`—which are connected by default to the keyboard, monitor, and monitor, and a file, respectively. A programmer can define additional streams.
- Streams are declared to be input, output, or both. A stream can be opened in the declaration or by a later call on `open()`. Opening a stream clears its status flags, creates a buffer, and attaches the stream



Fourth question is #1:

elem	.name	.symbol	.number
[0]	Sodium	Na	11
[1]	Argon	A	18
[2]	Aluminum	Al	13
[3]	Sulphur	S	16
[4]	Argon	A	18
[5]	Sulphur	S	16
[6]	Chlorine	Cl	17
[7]	Argon	A	18
[8]	??	??	?
	...	..	..
[19]	??	??	?

nq	many	k	score
8	4	1	3

question: Chlorine

After asking fourth question.

Fifth question is #3:

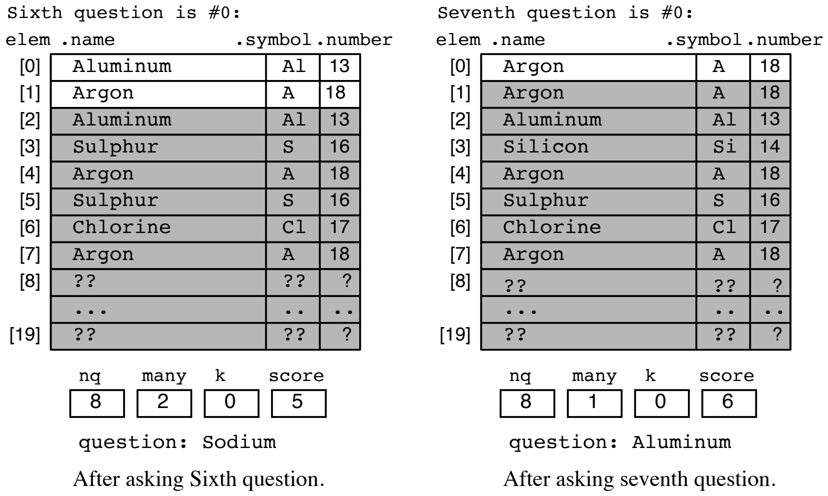
elem	.name	.symbol	.number
[0]	Sodium	Na	11
[1]	Argon	A	18
[2]	Aluminum	Al	13
[3]	Sulphur	S	16
[4]	Argon	A	18
[5]	Sulphur	S	16
[6]	Chlorine	Cl	17
[7]	Argon	A	18
[8]	??	??	?
	...	..	..
[19]	??	??	?

nq	many	k	score
8	3	3	4

question: Sulphur

After asking fifth question.

Figure 14.22. The array after five questions.



Sixth question is #0:

elem	.name	.symbol	.number
[0]	Aluminum	Al	13
[1]	Argon	A	18
[2]	Aluminum	Al	13
[3]	Sulphur	S	16
[4]	Argon	A	18
[5]	Sulphur	S	16
[6]	Chlorine	Cl	17
[7]	Argon	A	18
[8]	??	??	?
	...	..	..
[19]	??	??	?

nq	many	k	score
8	2	0	5

question: Sodium

After asking Sixth question.

Seventh question is #0:

elem	.name	.symbol	.number
[0]	Argon	A	18
[1]	Argon	A	18
[2]	Aluminum	Al	13
[3]	Silicon	Si	14
[4]	Argon	A	18
[5]	Sulphur	S	16
[6]	Chlorine	Cl	17
[7]	Argon	A	18
[8]	??	??	?
	...	..	..
[19]	??	??	?

nq	many	k	score
8	1	0	6

question: Aluminum

After asking seventh question.

Figure 14.23. The array after seven questions.

to some device that will supply or receive the data. A call on `close()` terminates the connection and sets the stream status to not-ready.

- Once open, a stream can be used to transfer either ASCII text or data in binary format. All of the I/O functions presented in this chapter operate in text mode. Binary I/O is introduced in Chapter 15.
- Except for `cerr`, streams are buffered. On input, bytes do not come directly from a device into the program. Rather, a block of data is transferred from the device into a buffer and waits there until the program is ready for it. Typically, it is read by the program in many smaller pieces. On output, bytes that are written go first into a buffer and stay there until the buffer is full or flushed for some other reason, such as an explicit call on `flush()`. If a program crashes, everything waiting in the output buffers is lost. The corresponding files are not closed and, therefore, not readable.
- File-streams can be read and written using the same operators and functions that operate on the standard streams: `>>` and `<<`, `getline(streamName, stringName)`, `streamName.getline( char buffer[])`, and `charVar = streamName.get()`.
- When reading data from a file, certain types of errors and exceptions are likely to occur:

  1. The `open()` function may fail to open the file properly.
  2. The end of the file will eventually be reached.
  3. An incorrect character may have been inserted into the file, polluting the data on that line.
  4. Data from a multipart data set may have been omitted, causing alignment errors.

  Methods and functions exist to detect and recover from these types of errors. A robust input routine was presented that incorporates detection and recovery methods.

New tools. These definitions from the tools library are constantly useful. You should familiarize yourself with them:

- The function `fatal()`is used like `printf()`. Call it to write an error message and a file name to the standard error stream after a file-processing error. It flushes all stream buffers, closes all files, and aborts execution properly.
- The function `cleanline()` reads and discards characters from the designated stream until a newline character is encountered. It was used in the quiz program, Figure 14.18.

14.7.2 Programming Style

Function follows form. When creating a class for a program, its form must follow that of the real-world object the program is modeling. Within the program, processing of all data should be delegated to the class that defines that data. Operations should be performed by calling a function in that class. This keeps the details of the representation separate from the main flow of program logic.

Modules should be designed to permit the programmer to think and talk about the problem on a conceptual level, not at a detailed level. For example, the quiz program in Figure 14.13 operates on an array of objects. It uses functions to call up a quiz (operating on the whole array), administer a quiz (operating on the whole array), and ask a single question (operating on a single item).

Opening files. Failure to be able to open a file is a common experience. This leads to the following maxim: Always check that the result of `open()` is a valid stream. Further, if a program uses files, all of them should all be opened "up front," even if they will not be used immediately. This avoids the situation in which human effort and computer time have been spent entering and processing data, only to find that it is impossible to open an output file to receive the results. The general principle is this: Acquire all necessary resources before committing human labor or other resources to a project.

Closing streams. All streams are closed automatically when the program exits. However, the `iostream` library provides the `close()` function to permit a stream to be closed before the end of the program. Closing an input file promptly releases it for use by another program. Closing an output file promptly protects it from loss of data due to possible program crashes later in execution. Closing a stream releases the memory space used by the buffers.

Flushing. The `flush()` function is defined in the C++ standard only for output streams. It is used automatically when a stream is closed. Normally, a programmer has no need to call it explicitly. The tools library defines `flush()` for input streams. This is helpful when an error occurs during interactive input.

End of file. The function `eof()` should be used to detect the end of an input file. This condition occurs when you attempt to read data that are not there, so test for end of file only *after* an input operation, not before it. This is the only way to distinguish normal eof condition from a stream error.

Exceptions and errors. File-based programs typically handle large amounts of prerecorded data. It is important for these programs to detect file input errors and inform the user of them. The input functions in the `iostream` library set error and status flags. These should be checked and appropriate action taken if it is important for a program to be robust. Also, an error log file can be kept. It should record as much information as possible about each input error, so that the user has some way to find the faulty data and correct them. The input routine developed in Figure **??** robustly handles many types of input errors. The order in which the error tests are made is important; use this example as a template in your programs.

14.7.3 Sticky Points and Common Errors

- Keep in mind the difference between a stream name and a file name. The file name identifies the data to the operating system. A stream name is used within a program. The `open()` function connects a stream to a file and the argument to `open()` is the only place the file name ever appears in a program.
- A common cause of disaster is to create an output file with the same name as an already existing file, which will "blow away" the earlier file. The most common "victims" of this error are the program's own source code or input file, whose name absentmindedly is typed in by the user. For this reason, we recommend using something distinctive, such as `.in` as part of every input file name and `.out` as part of every output file name.
- Another common mistake is fail to skip whitespace explicitly when using `getline()`, or to expect `>>` to read more than one word.
- An input file must be constructed correctly for the program that reads it. Most programs expect the last line of a file to end with a newline character.
- The first unprocessed character in an input stream's buffer often is the whitespace character that terminated the previous data item. Input formats must take this into account, especially when working with character or string input.

- Input errors, incorrectly formatted data, and end-of-file problems are frequently encountered exception conditions. Any program that is to be used by many people or for many months needs to test for and deal with these conditions robustly. Failure to do so can lead to diverse disasters, including machine crashes and erroneous program results.
- A nonnumeric character in a numeric field can throw a program into an infinite loop unless detected and dealt with. Minimally, the stream state must be cleared and a function such as `ignore(1)` must be called to remove the character that caused the error from the input stream. The function `cleanline()` removes the faulty character along with the rest of the current line.
- Use C++ strings when reading character data or entire lines of input. Do not use character arrays for this purpose. An array must be long enough to hold all of the characters that will be read, and that is totally unpredictable. If the array is too short, the extra characters will overlay something else in memory.

14.7.4 New and Revisited Vocabulary

These are the most important terms and concepts discussed in this chapter:

stream I/O
buffer
stream
file
binary file
text file
log file
status code
error flag
end of file
error detection
error recovery
stream manipulator
stream `getline()`
string `getline()`
robust program
buffered stream
unbuffered stream

The following keywords, constants, and C++ library functions are discussed in this chapter:

`ios::`
`iostream`
`ostream`
`istream`
`cin and cout`
`cerr`
`clog`
`open()`
`close()`
`iomanip`
`left and right`
`ws`
`scientific`
`fixed`
`setprecision()`
`setfill()`
`setw()`
`hex and dec`
`get()`
`getline()`
`<<`
`>>`
`cleanline()`
`fatal()`
`flush()`
`eof()`
`good()`

14.7.5 Where to Find More Information

- On the text website, a complete application for measuring torque is developed step-by-step.
- The file copy program in Figure 14.10 is reworked and simplified in Chapter 20. Interactive input is replaced by the use of command-line arguments.
- Binary-mode file input and output are also used in an image-processing application in Chapter 15.

Chapter 15

Calculating with Bits

Until now, we have used the basic numeric types and built complex data structures out of them. In this chapter, we look at numbers at a lower, more primitive level. We examine how bits are used to represent numbers and how those bits are interpreted in varied contexts. We present a variety of topics related to number representation and interpretation.

Most C and C++ programs use the familiar base-10 notation for both input and output. The programmers and users who work with these programs can forget that, inside the computer, all numbers are represented in **binary** (base 2), not **decimal** (base 10). Usually, we have no need to concern ourselves with the computer's internal representation.

Occasionally, though, a program must deal directly with hardware components or the actual pattern of bits stored in the memory. Applications such as cryptography, random number generators, and programs that control hardware switches and circuits may relate directly to the pattern of bits in a number, rather than to its numerical value. For these applications, base-10 notation is very inconvenient because it does not correspond in an easy way to the internal representation of numbers. It takes a little work, starting with a large base-10 number, to arrive at its binary representation. Throughout this chapter and its exercises, whenever a numerical value is used and it is ambiguous as to which number base is being used, that value will be subscripted. For example, 109_{10} means 109 in base 10, while 109_{16} means 109 in base 16.

Ordinary arithmetic operators are inadequate to work at the bit level because they deal with the values of numbers, not the patterns of bits. An entire additional set of operators is needed that operate on the bits themselves. In this chapter, we study the *hexadecimal notation*, (which is used with unsigned numbers), and the bitwise operators that C and C++ provides for low-level bit manipulation. Further details of the bit-level representation of signed and unsigned integers and algorithms for number-base conversion among bases 10, 2, and 16 are described in Appendix E.

Finally, we present *bitfield structures*, which provide a way to organize and give symbolic names to sets of bits, just as ordinary structures let us name sets of bytes.

15.1 Number Representation and Conversion

In this section, we explore the ways in which integers and floating-point numbers are represented in computers.

15.1.1 Hexadecimal Notation

We use **hexadecimal** notation to write machine addresses and to interface with certain hardware devices. Using hexadecimal notation is the easiest way to do some jobs because it translates directly to binary notation[1]. A programmer who wants to create a certain pattern of bits in the computer and cares about the pattern itself, not just the number it represents, should first write out the pattern in binary, then convert it to hexadecimal, and write a literal in the program. To properly use the hexadecimal notation, we need to store values in `unsigned int` or `unsigned long` variables. The following paragraphs describe the syntax for writing hexadecimal (hex) literals and performing input and output with hexadecimal values.

[1] The conversion method is given in Appendix **??** and examples can be seen in Figures 15.12, 15.14, 15.15, 15.16, and 15.17.

An integer literal can be written in either decimal or hexadecimal notation:

Decimal	Hex	Decimal	Hex	Decimal	Hex
0	0x0	16	0x10	65	0x41
7	0x7	17	0x11	91	0x5b
8	0x8	18	0x12	127	0x7f
10	0xA	30	0x1E	128	0x80
11	0xB	33	0x21	255	0xFF
15	0xF	64	0x40	32767	0x7FFF

Figure 15.1. Hexadecimal numeric literals.

Hexadecimal literals. Any integer or character can be written as either a decimal literal (base 10) or a **hexadecimal literal** (base 16).[2] As illustrated in Figure 15.1, a hex literal starts with the characters `0x` or `0X` (digit zero, letter ex). Hex digits from 10 to 15 are written with the letters `A...F` or `a...f`. Upper- and lower-case letters are acceptable for both uses and mean the same thing. A **hex character literal** is written as an escape character followed by the letter `x` and the character's hexadecimal code from the ASCII table; a few samples are given in Figure 15.9. These numeric codes should be used only if no symbolic form exists because they may not be **portable**, that is, they depend on the particular character code of the local computer and may not work on another kind of system. In contrast, quoted characters are portable.

15.1.2 Number Systems and Number Representation

Numbers are written using positional base notation; each digit in a number has a value equal to that digit times a place value, which is a power (positive or negative) of the base value. For example, the *decimal* (base 10) place values are the powers of 10. From the decimal point going left, these are $10^0 = 1$, $10^1 = 10$, $10^2 = 100$, $10^3 = 1{,}000$, and so forth. From the decimal point going right, these are $10^{-1} = 0.1$, $10^{-2} = 0.01$, $10^{-3} = 0.001$, $10^{-4} = 0.0001$, and so forth. Figure 15.2 shows the place values for *binary* (base 2) and *hexadecimal* (base 16); the chart shows those places that are relevant for a short integer.

Just as base-10 notation (decimal) uses 10 digits to represent numbers, hexadecimal uses 16 digits. The first 10 digits are 0–9; the last six are the letters A–F. Figure 15.6 shows the decimal values of the 16 hexadecimal digits; Figure E.7 shows some equivalent values in decimal and hexadecimal.

[2] Octal literals can be used, too, but their use is not as prevalent and so they are omitted from this text.

Place values for base 16 (hexadecimal) are shown on the left; base-2 (binary) place values are on the right. Each hexadecimal digit occupies the same memory space as four binary bits because $2^4 = 16$.

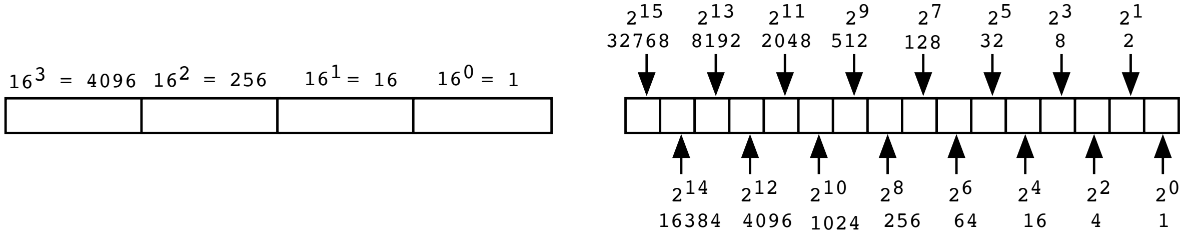


Figure 15.2. Place values.

The binary representations of several signed and unsigned integers follow. Several of these values turn up frequently during debugging, so it is useful to be able to recognize them.

$2^{15} = 32768$	$2^{14} = 16384$	$2^{13} = 8192$	$2^{12} = 4096$	$2^{11} = 2048$	$2^{10} = 1024$	$2^9 = 512$	$2^8 = 256$	$2^7 = 128$	$2^6 = 64$	$2^5 = 32$	$2^4 = 16$	$2^3 = 8$	$2^2 = 4$	$2^1 = 2$	$2^0 = 1$	Interpreted as a signed short int high order bit = -32768	Interpreted as an unsigned short int high order bit = 32768
0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	32767	32767
0	0	1	0	0	1	1	1	0	0	0	1	0	0	0	0	10000	10000
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	+1	+1
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	-32768 + 32767 = -1	+32768 + 32767 = 65535
1	1	0	1	1	0	0	0	1	1	1	1	0	0	0	0	-32768 + 22768 = -10000	+32768 + 22768 = 55536
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	-32768 + 0 = -32768	+32768 + 0 = 32768
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	-32768 + 1 = -32767	+32768 + 1 = 32769

Figure 15.3. Two's complement representation of integers.

15.1.3 Signed and Unsigned Integers

On paper or in a computer, all the bits in an unsigned number represent part of the number itself. Not so with a signed number: One bit must be used to represent the sign. The usual way that we represent a signed number on paper is by putting a positive or negative sign in front of a value of a given magnitude. This representation, called *sign and magnitude*, was used in early computers and still is used today for floating-point numbers. However, a different representation, called *two's complement*, is used for signed integers in most modern computers.

In this notation, the leftmost bit position has a negative place value for signed integers and a positive value for unsigned integers. All the rest of the bit positions have positive place values. Numbers with a 0 in the high-order position have the same interpretation whether they are signed or unsigned. However, a number with a 1 in the high-order position is negative when interpreted as a signed integer and a very large positive number when interpreted as unsigned. Several examples of positive and negative binary two's complement representations are shown in Figure 15.3.

To negate a number in two's complement, invert (complement) all of the bits and add 1 to the result. For example, the 16-bit binary representation of +27 is 00000000 00011011, so we find the representation of −27 by complementing these bits, 11111111 11100100, and then adding 1: 11111111 11100101.

You can tell whether a signed two's complement number is positive or negative by looking at the high-order bit; if it is 1, the number is negative. To find the magnitude of a negative integer, complement the bits and add 1. For example, suppose we are given the binary number 11111111 11010010. It is negative because it starts with a 1 bit. To find the magnitude we complement these bits, 00000000 00101101; add 1 using binary addition, and convert to its decimal form, 00000000 00101110 $= 32 + 8 + 4 + 2 = 46$. So the original number is −46.

Adding binary numbers is similar to adding decimal values, except that a carry is generated when the sum for a bit position is 2 or greater, rather than 10 or greater, as with decimal numbers. The carry values are represented in binary and may carry over into more than one position as they can in decimal addition.

Wrap revisited. Wrap is a representational error that happens when the result of a computation is too large to store in the target variable.

> **Issue:** Start with any positive integer. Keep adding 1. Eventually, the answer will be negative because of wrap. Start with any unsigned integer. Keep adding 1. Eventually, the answer will be 0 because of wrap.

With signed integers, wrap happens whenever there is a carry *into* the sign bit (leftmost bit). When x is the largest positive signed integer we can represent, $x+1$ will be the smallest (farthest from 0) negative integer.

The number -10 in binary IEEE format for type `float`:

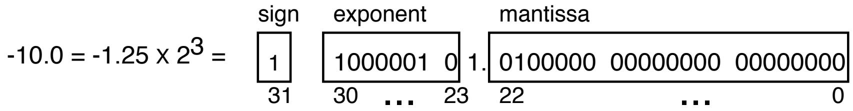


Figure 15.4. Binary representation of reals.

To be specific, for a 2-byte model, the largest signed value is 32,767, which is represented by the bit sequence x =01111111 11111111. The value of $x+1$ is 10000000 00000000 in binary and -32768 in base 10.

Similarly, with unsigned integers, wrap happens whenever there is a carry *out of* the leftmost bit of the integer. In this case, if x is the largest unsigned integer, $x+1$ will be 0. To be specific, for a 2-byte model, the largest unsigned value is 65,535, which is represented by the bit sequence x =11111111 11111111. The value, in binary, of $x+1$ is 00000000 00000000.

Wrap also can happen when any operation produces a result too large to store in the variable supposed to receive it. Unfortunately, there is no systematic way to detect wrap after it happens. Avoidance is the best policy, and that requires a combination of programmer awareness and caution when working with integers.

15.1.4 Representation of Real Numbers

A real number, N, is represented by a signed mantissa, m, multiplied by a base, b, raised to some signed exponent, x; that is,

$$N = \pm m \times b^{\pm x}$$

Inside the computer, each of the components of a real number is stored in some binary format. The IEEE (Institute for Electrical and Electronic Engineers) established a standard for floating-point representation and arithmetic that has been carefully designed to give predictable results with as much precision as possible. Most scientists doing serious numerical computations use systems that implement the IEEE standard. Figure E.3 shows the way that the number -10 is represented according to the IEEE standard for four-byte real numbers. This representation uses 32 bits divided into three fields to represent a real value. The base, $b = 2$, is not represented explicitly; it is built into the computer's floating-point hardware.

The mantissa is represented using the sign and magnitude format. The sign of the mantissa (which is also the sign of the number) is encoded in a single bit, bit 31, in a manner similar to that used for integers: A 0 for positive numbers or a 1 for negative values. The magnitude of the mantissa is separated from the sign, as indicated in Figure E.3, and occupies the right end of the number.

After every calculation, the mantissa of the result is *normalized*; that means it is returned to the form $1.XX\ldots X$, where each X represents a one or a zero. In this form, the leading bit always is 1 and is always followed by the decimal point. A number always is normalized before it is stored in a memory variable. Since all mantissas follow this rule, the 1 and the decimal point do not need to be stored explicitly; they are built into the hardware instead. Thus the 23 bits in the mantissa of an IEEE real number are used to store the 23 X bits.

In Figure 15.4, the fraction 0.25, or 0.01 in binary, is stored in the mantissa bits and the leading 1 is recreated by the hardware when the value is brought from memory into a register.

When we add or subtract real numbers on paper, the first step is to line up the decimal points of the numbers. A computer using floating-point arithmetic must start with a corresponding operation, denormalization. To add or subtract two numbers with different exponents, the bits in the mantissa of the operand with the smaller exponent must be **denormalized**: The mantissa bits are shifted rightward and the exponent is increased by 1 for each shifted bit position. The shifting process ends when the exponent equals the exponent of the larger operand. We call this number representation *floating point* because the computer hardware automatically "floats" the mantissa to the appropriate position for each addition or subtraction operation.

The precision of a floating-point number is the number of digits that are mathematically correct. Precision directly depends on the number of bits used to store the mantissa and the amount of error that may accumulate due to round-off during computations. Typically a calculation is performed using a few additional bits beyond the lengths of the original operands. The final result then must be rounded off to return it to the length of the

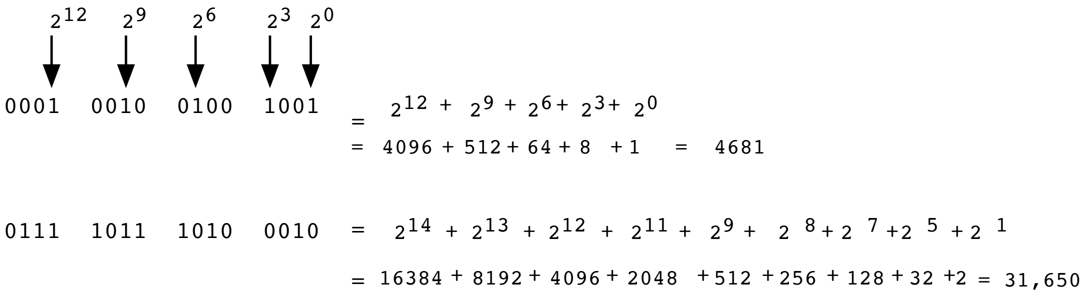


Figure 15.5. Converting binary numbers to decimal.

operands involved. The different floating-point types use different numbers of bits in the mantissa to achieve different levels of precision. Typical limits are given in Chapter 7.

The exponent is represented in bits 23 . . . 30, which are between the sign and the mantissa. Each time the mantissa is shifted right or left, the exponent is adjusted to preserve the value of the number. A shift of one bit position causes the exponent to be increased or decreased by 1. In the IEEE standard real format, the exponent is stored in *excess 127 notation*. Here, the entire 8 bits are treated as a positive value, then the excess value, 127, is subtracted from the 8-bit value to determine the true exponent. In Figure E.3, $127 + 3 = 130$, which is the value stored in the eight exponent bits. While this format may be complicated to understand, it has advantages in developing hardware to do quick comparisons and calculations with real numbers.

15.1.5 Base Conversion

Binary to decimal. We use the table of place values in Figure 15.2 when converting a number from base 2 to base 10. The process is simple and intuitive: Add the place values that correspond to the one bits in the binary representation. The result is the decimal representation (see Figure E.4).

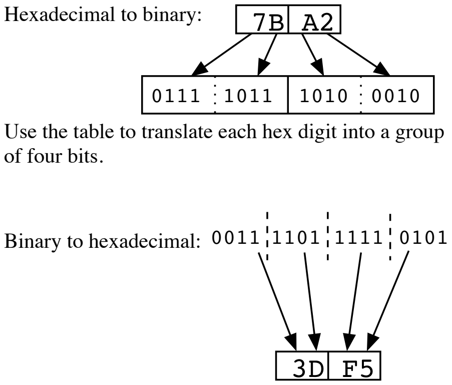
Binary to and from hexadecimal. When a programmer must work with numbers in binary or hexadecimal, it is useful to know how to go from one representation to the other. The binary and hexadecimal representations are closely related. Since $16 = 2^4$, each hex digit corresponds to 4 bits. Base conversion from hexadecimal to binary (or vice versa) is done by simply expanding (contracting) the number using the table in Figure E.5.

Decimal to binary. It also is possible to convert a base-10 number, N, into a base-2 number, T, using only the table of place values and a calculator or pencil and paper. First, look at the table in Figure E.1 and find the largest place value that is smaller than N. Subtract this value from N and write a 1 in the corresponding position in T. Keep the remainder for the next step in the process. Moving to the right in T, write a 1 if the next place value can be subtracted (subtract it and save the remainder) or a 0 otherwise. Continue this process, reducing the remainder of N until it becomes 0, then fill in all the remaining places of T with zeros. Figure E.6 illustrates this process of repeated subtraction for both an integer and a real value.

Hexadecimal to decimal. Converting a hexadecimal number to a decimal number is analogous to the binary-to-decimal conversion. Each digit of the hexadecimal representation must be converted to its decimal value (for example, C and F must be converted to 12 and 15, respectively). Then each digit's decimal value must be multiplied by its place value and the results added. This process is illustrated in Figure E.7.

Decimal to hexadecimal. The easiest way to convert a number from base 10 to base 16 is first to convert the number to binary, then convert the result to base 16. The job also can be done by dividing the number repeatedly by 16; the remainder on each division, when converted to a hex digit, becomes the next digit of the answer, going right to left. We do not recommend this method, however, because it is difficult to do in your head and awkward to calculate remainders on most pocket calculators.

Decimal	Hex	Binary
0	0	0000
1	1	0001
2	2	0010
3	3	0011
4	4	0100
5	5	0101
6	6	0110
7	7	0111
8	8	1000
9	9	1001
10	A	1010
11	B	1011
12	C	1100
13	D	1101
14	E	1110
15	F	1111



Use the table to translate each hex digit into a group of four bits.

Mark off the bits into groups of four, starting at the right. Add leading zeros on the left, if you wish, to make an even group of four bits. Translate each group to one hex digit, using the table.

Hexadecimal to Binary Conversion					
hex	dec	binary	hex	dec	binary
0	0	0000	8	8	1000
1	1	0001	9	9	1001
2	2	0010	A	10	1010
3	3	0011	B	11	1011
4	4	0100	C	12	1100
5	5	0101	D	13	1101
6	6	0110	E	14	1110
7	7	0111	F	15	1111

Figure 15.6. Converting between hexadecimal and binary.

$$\begin{array}{rl}
10{,}542 & = 10\ 1001\ 1000\ 1110 \\
\underline{-8{,}096} & = 2^{13} \\
2{,}446 & \\
\underline{-2{,}048} & = 2^{11} \\
398 & \\
\underline{-\ \ 256} & = 2^{8} \\
142 & \\
\underline{-\ \ 128} & = 2^{7} \\
14 & \\
\underline{-\ \ \ \ 8} & = 2^{3} \\
6 & \\
\underline{-\ \ \ \ 4} & = 2^{2} \\
2 & \\
\underline{-\ \ \ \ 2} & = 2^{1} \\
0 &
\end{array}
\qquad
\begin{array}{rl}
630.3125 & = 10\ 0111\ 0110\ .\ 0101 \\
\underline{-512.\qquad} & = 2^{9} \\
118.3125 & \\
\underline{-\ 64.\qquad} & = 2^{6} \\
54.3125 & \\
\underline{-\ 32.\qquad} & = 2^{5} \\
22.3125 & \\
\underline{-\ 16.\qquad} & = 2^{4} \\
6.3125 & \\
\underline{-\ \ 4.\qquad} & = 2^{2} \\
2.3125 & \\
\underline{-\ \ 2.\qquad} & = 2^{1} \\
0.3125 & \\
\underline{-\ \ 0.25\ \ } & = 2^{-2} \\
0.0625 & \\
\underline{-\ \ 0.0625} & = 2^{-4} \\
0 &
\end{array}$$

Figure 15.7. Converting decimal numbers to binary.

```
2A3C = 2*16^3 + 10*16^2 + 3*16 + 12
     = 2*4096 + 10*256 + 48 + 12 = 10,812
BFD = 11*16^2 + 15*16 + 13
    = 11*256 + 240 + 13 = 2,816 + 253 = 3,069
A2.D2 = 10*16^1 + 2*16^0 + 13*16^-1 + 2*16^-2
      = 160 + 2 + .8125 + 0.0078125 = 162.8203125
```

Figure 15.8. Converting hexadecimal numbers to decimal.

Character literals can be written symbolically as an escape character followed by the letter `x` and the character's hexadecimal code from the ASCII table:

Meaning	Symbol (portable)	Hex Escape Code (ASCII only)
The letter A	`'A'`	`'\x41'`
The letter a	`'a'`	`'\x61'`
Newline	`'\n'`	`'\xA'`
Formfeed	`'\f'`	`'\xC'`
Blank space	`' '`	`'\x20'`
Escape character	`'\\'`	`'\x1B'`
Null character	`'\0'`	`'\x00'`

Figure 15.9. Hexadecimal character literals.

15.2 Hexadecimal Input and Output

Input and output in binary notation are not directly supported in C++ because binary numbers are difficult for human beings to read and use accurately. We do not work well with rows of bits; they make our eyes swim. Instead, all input and output is done in decimal, octal, or hexadecimal notation. When we use `>>` to read input into an integer variable, the type of the variable tells the system to read an integer, and the state of the stream flags tells whether that will be converted to binary using base 10 or base 16. All numbers are stored in binary. The default stream setting is to use base 10, and the stream manipulators `hex`, `dec` and `oct` (for octal) are used to change the status flags.

The opposite conversion (binary to a string of digits) is done by `<<` when we output a number. These conversions are done automatically, and we often forget they happen. But on some occasions, base-10 representation is not convenient for input or is confusing for output. For example, if a programmer needs to write a mask value or a disk address, the value should be written as an unsigned integer and it is often easiest to work with hexadecimal form.

Reading and writing integers in hexadecimal notation. The program in Figure **??** demonstrates how to read and print unsigned integers using both decimal and hex notations. By showing numbers in both base 10 and base 16, we hope to build some understanding of the numbers and their representations.

Notes on Figure 15.10: I/O for hex and decimal integers.

First box: Input in base 10.

- The input for an unsigned integer is expected to be a sequence of decimal digits with no leading sign.
- The syntax is the same for reading a `short int`, `an int`, and a `long int`.
- The input variable's type determines what happens; if the number that is read is too large to store in the variable, the maximum value for the variable's type is stored instead.
- Whether the input was in decimal or hex notation, and whether the hex digits were entered in upper or lower case, the output will be printed according to whether the stream is in `hex` or `dec` or `oct` mode.
- Warning: as is shown here, if you use `hex` or `oct`, *always put the stream back into* `dec` *mode.*
- Note that a number bigger than 7 *looks* larger printed in octal than in decimal, and larger in decimal than in hex.
- A sample output from this box is:

```
Please enter an unsigned int: 345
   = 345 in decimal and = 159 in hex, and 531 in octal.
```

- Here is an input that is too large to store in the variable. When this happens, the stream's `fail` flag is turned on and the variable is set to the maximum value for its type. Note the ffff printed in hex – that is the maximum number that can be stored in a `short int`.

```
Please enter an unsigned int: 66000
   = 65535 in decimal and = ffff in hex, and 177777 in octal.
```

- When a negative number is entered, instead of an unsigned number, the stream's `fail` flag is turned on and the variable is set to 0, as in the following output:

This program demonstrates how unsigned integers of various sizes may be read and written.

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

int main( void )
{
    unsigned short ui, xi;

    cout <<"\n Please enter an unsigned int:  ";
    cin >>dec >>ui;
    cout <<" = " <<ui <<" in decimal and = "
         <<hex <<ui <<" in hex, and = "
         <<oct <<ui <<" in octal.\n" <<dec;

    cout <<"\n Please enter an unsigned int in hex:  ";
    cin >>hex >>xi ;
    cout <<" = " <<xi <<" in decimal and = "
         <<hex <<xi <<" in hex, and = "
         <<oct <<xi <<" in octal.\n\n" <<dec;
    return 0;
}
```

Figure 15.10. I/O for hex and decimal integers.

```
Please enter an unsigned int: -31
   = 0 in decimal and = 0 in hex.
```

Second box: Input in hexadecimal.

- To cause input to be in hexadecimal, set the input stream to `>>hex`.
- Hexadecimal input may include any mixture of digits from 1 to 9 and letters from a to f or from A to F.

```
Please enter an unsigned int in hex: aBc
  = 2748 in decimal and = abc in hex, and = 5274 in octal.
```

- The input may have a leading `0x` to indicate hex, but it is also acceptable to omit the `0x`:

```
Please enter an unsigned int in hex: 0x12AB
   = 4779 in decimal and = 12ab in hex, and = 11253 in octal.
```

15.3 Bitwise Operators

C and C++ include six operators whose purpose is to manipulate the bits inside a byte. For most applications, we do not need these operators. However, systems programmers use them extensively when implementing type conversions and hardware interfaces. Any program that must deal directly with a hardware device may need to pack bits into the instruction format for that device or unpack status words returned by the device. Another application is an archive program that compresses the data in a file so that it takes up less space as an archive file. A program that works with the DES encryption algorithm makes extensive use of bitwise operators to encode or decode information.

Bitwise operators fall into two categories: shift operators move the bits toward the left or the right within a byte, and **bitwise-logic** operators can turn individual bits or groups of bits on or off or **toggle** them. These operators are listed in Figure 15.11. The operands of all these operators must be one of the integer types (type `unsigned` or `int` and length `long`, `short`, or `char`). Because we cannot use the subscript operator to reference bits individually, we must use some combination of the bitwise operators to extract or change the value of a given bit in a byte.

Arity	Symbol	Meaning	Precedence	Use and Result
Unary	∼	Bitwise complement	15	Reverse all bits
Binary	<<	Left shift	11	Move bits left
	>>	Right shift	11	Move bits right
	&	Bitwise AND	8	Turn bits off or decompose
	∧	Bitwise exclusive OR (XOR)	7	Toggle bits
	|	Bitwise OR	6	Turn bits on or recombine

Figure 15.11. Bitwise operators.

15.3.1 Masks and Masking

When a program manipulates codes or uses hardware bit switches, it often must isolate one or more bits from the other bits that are stored in the same byte. The process used to do this job is called *masking*. In a masking operation, we use a bit operator and a constant called a **mask** that contains a bit pattern with a 1 bit corresponding to each position of a bit that must be isolated and a 0 bit in every other position.

For example, suppose we want to encode a file so that our competitors cannot understand it (a process called *encryption*). As part of this process, we want to split a short integer into four portions, A (3 bits), B (5 bits), C (4 bits), and D (4 bits), and put them back together in the scrambled order: C, A, D, B. We start by writing out the bit patterns, in binary and hex, for isolating the four portions. The hex version then goes into a `#define` statement. The table in Figure 15.12 shows these patterns. Then we use the `&` operator with each mask to decompose the data, shift instructions to move the pieces around, and `|` operations to put them back together in the new order. This simple technique for **encoding and decoding** is illustrated in the next section in Figures 15.22 and 15.23.

Bitwise AND (`&`) turns bits off. The bitwise AND operator, `&`, is defined by the fourth column of the truth table shown in Figure 15.13, which is the same as the truth table for logical AND. The difference is that logical operators are applied to the truth values of their operands, and bitwise operators are applied to each pair of corresponding bits in the operands. Figure 15.14 shows three examples of the use of bitwise AND. To get the answer (on the bottom line) look at each pair of corresponding bits from the two operands above it, and apply the bitwise AND truth table to them.

The bitwise AND operator can be used with a mask to isolate a bit field by "turning off" all the other bits in the result. (None of the bitwise operators affect the value of the operand in memory.) The first column in Figure 15.14 shows that using a mask with 0 bits will turn off the bits corresponding to those zeros. The second column shows how a mask with a field of four 1 bits can be used to isolate the corresponding field of `x`. The third column of this figure illustrates the semantics of `&` by pairing up all combinations of 1 and 0 bits.

Part	Bit Pattern	`#define`, with Hex Constant
A	11100000 00000000	`#define A  0xE000`
B	00011111 00000000	`#define B  0x1F00`
C	00000000 11110000	`#define C  0x00F0`
D	00000000 00001111	`#define D  0x000F`

Figure 15.12. Bit masks for encrypting a number.

x	y	∼x	x & y	x | y	x ∧ y
0	0	1	0	0	0
0	1	1	0	1	1
1	0	0	0	1	1
1	1	0	1	1	0

Figure 15.13. Truth tables for bitwise operators.

The bit vectors used here are 1 byte long. A result bit is 1 if both operand bits are 1.

	Binary	Hex		Binary	Hex		Binary	Hex
x	1111 1111	0xFF	x	0111 1010	0x7A	x	0111 1010	0x7A
mask	0010 0000	0x20	mask	1111 0000	0xF0	y	1010 0011	0xA3
x & mask	0010 0000	0x20	x & mask	0111 0000	0x70	x & y	0010 0010	0x22

Figure 15.14. Bitwise AND (&).

Note that a bit in the result is 1 only if both of the corresponding bits in the operands are 1.

Bitwise OR (|) turns bits on. Just as the bitwise AND operator can be used to turn bits off or decompose a bit vector, the bitwise OR operator, defined in the fifth column of Figure 15.13, can be used to turn on bits or reassemble the parts of a bit vector. Figure 15.15 shows three examples of the use of this operator. The first column shows how a mask can be used with | to turn on a single bit. The second column shows how two fields in different bit vectors can be combined into one vector with bitwise OR. The third column of this figure illustrates the semantics of | by pairing up all combinations of 1 and 0 bits. Note that a bit in the result is 1 if either of the corresponding bits is 1.

Complement (∼) and bitwise XOR (∧) toggle bits. Sometimes we want to turn a bit on, sometimes off, and sometimes we want to simply change it, whatever its current state is. For example, if we are implementing a keyboard handler, we need to change from upper-case to lower-case or vice versa if the user presses the Caps Lock key. When we change a switch setting like this, we say we *toggle* the switch. The complement operator, ˜, defined by the third column in Figure 15.13, toggles all the bits in a word, turning 1's to 0's and 0's to 1's.

The bitwise XOR operator (∼) can be used with a mask to toggle some of the bits in a vector and leave the rest unchanged. This is essential when several switches are packed into the same control word. The first two columns of Figure 15.16 show how to use XOR with a mask to toggle some of the bits in a vector but leave others unaffected. Of course, to actually change the switch setting, you must use = to store the result back into the switch variable. The last column of this figure illustrates the semantics of ∼ by pairing up all combinations of 1 and 0 bits. Note that a bit in the result is 1 if the corresponding bits in the operands are *different.*

The three negations. C has three unary operators that are alike but different. Figure 15.17 shows the operation of these operators: logical NOT, complement, and arithmetic negation. All of them compute some kind of opposite, but on most machines, these three "opposites" are different values. Normally, this is no problem, since you use the three operators for different types of operands and in different situations. This discussion is included for those who are curious about why we need three ways to say no.

The bit vectors used here are 1 byte long. A result bit is 1 if either operand bit is 1.

	Binary	Hex		Binary	Hex		Binary	Hex
x	0000 0000	0x00	x	0000 1100	0x0C	x	0011 1110	0x3E
mask	0010 0000	0x20	mask	1111 0000	0xF0	y	1010 0111	0xA7
x | mask	0010 0000	0x20	x | mask	1111 1100	0xFC	x | y	1011 1111	0xBF

Figure 15.15. Bitwise OR (|).

The bit vectors used here are 1 byte long. A result bit is 1 if the operand bits are different.

	Binary	Hex		Binary	Hex		Binary	Hex
x	1111 1111	0xFF	x	1100 1100	0xCC	x	0111 1010	0x7A
mask	0011 0000	0x30	mask	1111 0000	0xF0	y	1010 0011	0xA3
x ∧ mask	1100 1111	0xCF	x ∧ mask	0011 1100	0x3C	x ∧ y	1101 1001	0xD9

Figure 15.16. Bitwise XOR (∧).

We illustrate the results of the three ways to say no: logical NOT, complement, and negate. The bit vectors used here are 1 byte long, the representation for negative numbers is two's complement.

	Decimal	Hex	Binary		Decimal	Hex	Binary		Decimal	Hex	Binary
`x`	0	`0x00`	0000 0000	`x`	1	`0x01`	0000 0001	`x`	−10	`0xF6`	1111 0110
`!x`	1	`0x01`	0000 0001	`!x`	0	`0x00`	0000 0000	`!x`	0	`0x00`	0000 0000
`~x`	−1	`0xFF`	1111 1111	`~x`	−2	`0xFE`	1111 1110	`~x`	9	`0x09`	0000 1001
`-x`	0	`0x00`	0000 0000	`-x`	−1	`0xFF`	1111 1111	`-x`	10	`0x0A`	0000 1010

Figure 15.17. Just say no.

The logical NOT operator (`!`) is the simplest (see the second row in Figure 15.17). It turns true values to `false` and false values to `true`. True is represented by the *integer 1*, which has many 0 bits and only a single 1 bit.

The complement operator, which toggles all the bits, is shown on the third row in Figure 15.17. Looking at the first column of Figure 15.17, note how the result of `!x` is very different from the result of `~x`.

Most modern computers use two's complement notation to represent negative integers. The **two's complement** of a value is always 1 greater than the bitwise complement (also known as one's complement) of that number.[3] We can see this relationship by comparing the third and fourth lines in Figure 15.17.

15.3.2 Shift Operators

Once we have used `&` to decompose a bit vector or isolate a switch setting, we generally need to move the bits from the middle of the result to the end before further processing. Conversely, to set a switch or recompose a word, bits usually need to be shifted from the right end of a word to the appropriate position before using `|` to recombine them. This is the purpose of the right-shift (`>>`) and left-shift (`<<`) operators.

The left operand of a shift operator can be either a signed or unsigned integer. This is the bit vector that is shifted. The right operand must be a nonnegative integer;[4] it is the number of times that the vector will be shifted by one bit position. Figure 15.18 shows examples of both shift operations.

Left shifts. Left shifts move the bits of the vector to the left; bits that move off the left end are forgotten, and 0 bits are pulled in to fill the right end. This is a straightforward operation and very fast for hardware. Shifting left by one position has the same effect on a number as multiplying by 2 but it is a lot faster for the machine. In the table, you can see that the result of `n << 2` is four times the value of `n`, as long as no significant bits fall off the left end.

[3] This is the definition of *two's complement.* Appendix E explains this topic in more detail.

[4] The `C` standard specifies that negative shift amounts are undefined.

The table uses 1-byte variables declared thus:

```
signed   char s;      // A one-byte signed integer.
unsigned char u;      // A one-byte unsigned integer.
```

Signed	Decimal	Hex	Binary	Unsigned	Decimal	Hex	Binary
`s`	15	`0x0F`	0000 1111	`u`	10	`0x0A`	0000 1010
`s << 2`	60	`0x3C`	0011 1100	`u << 2`	40	`0x28`	0010 1000
`s >> 2`	3	`0x03`	0000 0011	`u >> 2`	2	`0x02`	0000 0010
`s`	−10	`0xF6`	1111 0110	`u`	255	`0xFF`	1111 1111
`s << 2`	−40	`0xD8`	1101 1000	`u << 2`	252	`0xFC`	1111 1100
`s >> 2`	−3	`0xFD`	1111 1101	`u >> 2`	63	`0x3F`	0011 1111

Figure 15.18. Left and right shifts.

Problem scope: Read a 32-bit Internet IP address in the form used to store addresses internally and print it in the four-part dotted form that we customarily see.

Input: A 32-bit integer in hex. For example: `fa1254b9`

Output required: The dotted form of the address. For the example given, this is: 250.18.84.185

Figure 15.19. Problem specifications: Decoding an Internet address.

One thing to remember is that a shift operation does not change the number of bits in a vector. If you start with a 16-bit vector and shift it 10 places to the left, the result is a 16-bit vector that has lost its 10 high-order bits and has had 10 0 bits inserted on the right end.

Right shifts. Unfortunately, right shifts are not as simple as left shifts. This is because computer hardware typically supports two kinds of right shifts, signed right shifts and unsigned right shifts. In C, an **unsigned shift** always is used if the operand is declared to be unsigned. A **signed shift** is used for signed operands.[5]

A signed right shift fills the left end with copies of the sign bit (the leftmost bit) as the number is shifted, and an unsigned shift fills with 0 bits. The reason for having signed shifts at all is so that the operation can be used in arithmetic processing; with a signed shift, the sign of a number will not be changed from negative to positive when it is shifted. There is no difference between the effects of signed and unsigned shifts for positive numbers, since all have a 0 bit in the leftmost position anyway.

Analogous to left shift, a right shift by one position divides a number by 2 (and discards the remainder). A right shift by n positions divides a number by 2^n. In the table, you can see that `15 >> 2` gives the same result as 15/4.

15.3.3 Example: Shifting and Masking an Internet Address

IP (Internet protocol) addresses normally are written as four small integers separated by dots, as in `32.55.1.102`. The machine representation of these addresses is an unsigned long integer. Figure 15.19 lists the specifications for decoding an IP address. The program in Figure 15.20 takes the hex representation of an address, decodes it, and prints it in the four-part dotted form we are accustomed to reading.

Notes on Figure 15.20: Decoding an Internet address.

First box: the mask. We define a mask to isolate the last 8 bits of a long integer. We can write the constant with or without leading zeros, as `0xffL` or `0x000000ffL`, but we need the `L` to make it a long integer.

Second box: the variables.

- We declare the input variable as `unsigned` so that unsigned shift operations will be used and `long` because the input has 32 bits.
- We use each of the other four variables to hold one field of the IP address as we decompose it. These could be short integers since the values are in the range 0. . . 255.

Third box: hex input and output.

- We put the input stream into hex mode before reading the input. When done, we put it back into decimal mode.
- To format the output so that all 8 hex digits are displayed, it is necessary to set a field width of 8 columns and to set the fill character to '0'. The fill character will stay set until explicitly changed. The field width must be set separately for each thing printed.

Fourth box: byte decomposition.

- The coded address contains four fields; each is 8 bits. The byte mask lets us isolate the rightmost 8 bits in an address. In this box, we store each successive 8-bit field of the input into one of the `f` variables.
- To get `f1`, the first field of the IP address, we shift the address 24 places to the right so that all but the first 8 bits are discarded.

[5] According to the standard, an implementor could choose to use an unsigned shift in both cases. However, signed shifts normally are used for signed values.

The problem specifications are in Figure 15.19.

```
#include <iostream>
using namespace std;
#define  BYTEMASK  0xffL        The L is to make a long integer.

int main( void )
{
    unsigned long ipAddress;
    unsigned f1, f2, f3, f4;

    cout <<"\n Please enter an IP address as 8 hex digits:  ";
    cin >>hex >>ipAddress >>dec;
    cout <<"\t You have entered " <<setw(8) <<setfill('0')
         <<hex <<ipAddress <<endl;

    f1 = ipAddress >>  24 & BYTEMASK;
    f2 = ipAddress >>  16 & BYTEMASK;
    f3 = ipAddress >>   8 & BYTEMASK;
    f4 = ipAddress        & BYTEMASK;

    cout <<dec <<"\t The IP address in standard form is:  "
         <<f1 <<"." <<f2 <<"." <<f3 <<"." <<f4 <<endl;
}
```

Figure 15.20. Decoding an Internet address.

- If all machines used 32-bit long integers, we would not need the masking operation to compute `f1`. We include it to be sure that this code can be used on machines with longer integers.
- To get `f2` and `f3`, we shift the input by smaller amounts and mask out everything except the 8 bits on the right.
- Since `f4` is supposed to be the rightmost 8 bits of the IP address, we can mask the input directly, without shifting.

The output from two sample runs is

```
Please enter an IP address as 8 hex digits: fa1254b9
        You have entered fa1254b9
        The IP address in standard form is: 250.18.84.185
----------------------------------------------------
 Please enter an IP address as 8 hex digits: 0023fa01
        You have entered 0023fa01
        The IP address in standard form is: 0.35.250.1
```

15.4 Application: Simple Encryption and Decryption

We have enough tools to write a simple application program that uses the bitwise operators. Let us return to the encryption example[6]. Using the masks in Figure 15.12, we encrypt a short integer by breaking it into four portions, A (3 bits), B (5 bits), C (4 bits), and D (4 bits), and putting them back together in a scrambled order: C, A, D, B. That is, we start with the bits in this order: `aaabbbbbccccdddd` and end with the scrambled order: `ccccaaaddddbbbbb`. Figure 15.21 specifies the problem, Figure 15.22 is the encryption program, and

[6] Please note that this is just an programming example; it is not good cryptography. Nonprofessionals should never try to invent cryptosystems – the result will not be secure.

Problem scope: Write a function to encrypt a short integer and a main program to test it. Write a matching decryption program.

Input: Any short integer.

Output required: The input will be echoed in both decimal and hex; the encrypted number also will be given in both bases.

Formula: Divide the 16 bits into four unequal-length fields (3, 5, 4, and 4 bits) and rearrange those fields within the space as shown:

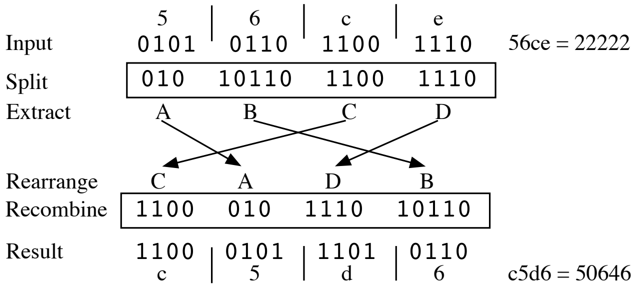


Figure 15.21. Problem specifications: Encrypting and decrypting a number.

Figure 15.23 is the program for subsequent decryption. The encrypted results of this program might fool your friends, but it is not a secure system.

Notes on Figures 15.22 and 15.23. Encrypting and decrypting a number.

First boxes: the masks.

- We refer to the bit positions with numbers 0. . . 15, where bit 15 is the leftmost and bit 0 is the last bit on the right.
- The diagram in Figure 15.21 shows the positions of the four fields and the the names that are used in the program for those positions.
- We define one bit mask for each part of the number we want to isolate. The hex constants for encryption are developed in Figure 15.12.
- The bit masks for the decryption process allow us to isolate the same four fields in different positions within the word so that we can reverse the encryption process.

Second boxes: the interfaces for functions `encrypt()` and `decrypt()`.

- We use type `unsigned` for all bit manipulation, so that the leftmost bit is treated just like any other bit. (With signed types, the leftmost bit is treated differently because it is the sign.)
- We are working with 16-bit numbers, so we use type `short`. Therefore, the argument type and the return type are both `unsigned short`.
- When a signed short integer is cast to an unsigned short integer, or vice versa, no bits change within the word. This is guaranteed by the language standard for the cast from signed to unsigned. It is officially "undefined" for casting from unsigned to signed. However, on a 2's complement machine, the result will almost certainly be no change in the bit pattern. Thus, if we cast a signed int to an unsigned, then back to signed int, the result will be the same number we started with.

Third boxes: unpacking.

- The `&` operation turns off all the bits of `n` except those that correspond to the 1 bits in the mask. This lets us isolate one bitfield so we can store it in a separate variable. For example, here we start with 22222 = 56ce and apply the b-mask to get field b of the word:

22222 = 56ce:	0101	0110	1100	1110
& BE = 0x1f:	0001	1111	0000	0000
field b:	0001	0110	0000	0000

- We use four `&` expressions, one for each segment of the data. We call this process **unpacking** the fields.
- After masking, we shift the isolated bits to their new positions. To compute the shift amount, we subtract the original position of the first bit in the field from its new position. We use `>>` for positive shift amounts, `<<` for negative amounts.

before >> 8	0001	0110	0000	0000
after >> 8	0000	0000	0001	0110

This program carries out the first half of the specifications in Figure 15.21.

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

#define AE  0xE000        // bits 15:13 end up as 11:9
#define BE  0x1F00        // bits 12:8  end up as  4:0
#define CE  0x00F0        // bits  7:4  end up as 15:12
#define DE  0x000F        // bits  3:0  end up as  8:5

unsigned short encrypt( unsigned short n );

int main( void )
{
    short in;
    unsigned short crypt;

    cout <<"Enter a short number to encrypt:  ";
    cin >>in;

    // Cast the int to unsigned before calling encrypt.
    crypt = encrypt( (unsigned short) in );

    cout <<"\nThe input number in base 10 is:  " <<in <<" \n"
         <<"The input number in hexadecimal is:  " <<hex <<in <<" \n\n"
         <<"The encrypted number in base 10 is:  " <<dec <<crypt <<"\n"
         <<"The encrypted number in base 16 is:  " <<hex <<crypt <<"\n\n";
}

// --------------------------------------------------------------------
unsigned short
encrypt( unsigned short n )
{
    unsigned short a, b, c, d;              // for the four parts of n

    a = (n & AE) >> 4;   // Isolate bits 15:13, shift to positions 11:9.
    b = (n & BE) >> 8;   // Isolate bits 12:8,  shift to positions 4:0.
    c = (n & CE) << 8;   // Isolate bits  7:4,  shift to positions 15:12.
    d = (n & DE) << 5;   // Isolate bits  3:0,  shift to positions 8:5.

    return c | a | d | b ;      // Pack the four parts back together.
}
```

Figure 15.22. Encrypting a number.

- We use four mask-and-shift expressions, one for each segment of the data.

Fourth boxes: packing.

- We use the | operator to put the four parts back together. We call this process *packing*.

```
     a :  0000  0100  0000  0000
     b :  0000  0000  0001  0110
     c :  1100  0000  0000  0000
|    d :  0000  0001  1100  0000
-------------------------------------
Result:   1100  0101  1101  0110    c5d6 = 50646
```

- It does not matter in what order the operands are listed in the | expression, we get the same result whether we write `a | b | c | d` *or* `c | a | d | b`.

Decryption is the inverse of encryption. This program looks very much the same as the encryption program in Figure 15.22 except that every operation is reversed.

```
#include <stdio.h>
#define AD 0x0E00         // bits 11:9  goto 15:13
#define BD 0x001F         // bits  4:0  goto 12:8
#define CD 0xF000         // bits 15:12 goto 7:4
#define DD 0x01E0         // bits  8:5  goto 3:0

unsigned short decrypt( unsigned short n );

int main( void )
{
    unsigned short in;
    short decrypted;;

    cout <<"Enter an encrypted short int in hex:  ";
    cin >>hex >>in;
    decrypted = (signed short)decrypt( in );

    cout <<"\nThe input number in base 10 is:  " <<in <<"\n"
    <<"The input number in hexadecimal is:  " <<hex <<in <<" \n\n"
    <<"The decrypted number in base 10 is:  " <<dec <<decrypted <<" \n"
    <<"The decrypted number in base 16 is:  " <<hex <<decrypted <<" \n\n";
    return 0; }
    // ---------------------------------------------------------------------
    unsigned short                     // use unsigned for bit manipulation
    decrypt( unsigned short n)
    {
        unsigned short a, b, c, d;             // for the four parts of n

        a = (n & AD) << 4;// Isolate bits 4:6, shift to positions 0:2.
        b = (n & BD) << 8;// Isolate bits 11:15, shift to positions 3:7.
        c = (n & CD) >> 8;// Isolate bits 0:3, shift to positions 8:11.
        d = (n & DD) >> 5;// Isolate bits 7:10, shift to positions 12:15.

        return a | b | c | d ;      // Pack the four parts back together.
    }
```

Figure 15.23. Decrypting a number.

Encryption.

- Here we encrypt the number shown in the diagrams:

  ```
  Enter a short number to encrypt: 22222

  The input number in base 10 is:     22222
  The bits of the input number are:   56ce

  The encrypted number in base 10 is: 50646
  The encrypted number in base 16 is: c5d6
  ```

- This encryption process shuffles the bits in a word without changing any of them. That is what makes it easy to reverse and decrypt. If all the bits are the same, the encrypted value will be exactly like the original value. So if the input number is -1, its hex representation is `ffff`, and the encrypted value is also `ffff`. Similarly 0 (or 0000) gets transformed to `0000`.
- Here is a negative number:

  ```
  Enter a short number to encrypt: -22222

  The input number in base 10 is:     -22222
  The bits of the input number are:   a932

  The encrypted number in base 10 is: 14921
  The encrypted number in base 16 is: 3a49
  ```

Decryption.

- The output from decryption of the second example is

  ```
  Enter an encrypted short int in hex: 3a49

  The input number in base 10 is:   14921
  The input number in hexadecimal is: 3a49

  The decrypted number in base 10 is: -22222
  The decrypted number in base 16 is: a932
  ```

15.5 Bitfield Types

A bitfield declaration defines a structured type in which the fields are mapped onto groups of bits rather than groups of bytes. **Bitfield structure** types are useful in programs that interface with and manipulate hardware devices, turning on and turning off individual bits in specific positions at fixed memory addresses. The bitfield declaration lets us represent and manipulate such devices symbolically, an important aid to processing them correctly.

A bitfield declaration has the same elements as any `struct` declaration, with the restriction that the base type of each field is an unsigned integer (char, short, int, or enum) and there is a colon and an integer (field width, in bits) after each field name. The field name is optional. Fields can be any width, and the total size of a bitfield structure may exceed one byte. The run-time system packs and unpacks the fields for you, enabling the programmer to use the names of the fields without concern for where or how they are stored.

The precise rules and restrictions for bitfield declarations are implementation dependent, as is the order in which the fields will be laid out in memory. This is unavoidable: computer architecture is not at all standardized and dealing with memory below the byte level obviously involves the nonstandard aspects of the hardware.

A big endian computer stores the high-order byte of an `int` at the lowest memory address and stores the bytes in order with the low-order end of the number at the highest memory address. A little endian machine stores bytes in memory in exactly the opposite order! In both kinds of machines, numbers in the CPU's registers are stored in big-endian order. Thus, anything that relies on byte order in a computer is only portable to other machines of the same basic architecture. The current Intel machines are little-endian.

The program in Figure 15.24 can be used to determine how your own hardware and compiler treat bitfields. It declares a bitfield object, initializes it, prints various information about it, and returns a hexadecimal dump of the contents so that the placement of information within the object can be determined. The packing diagram at the bottom of the figure shows the way this structure is implemented by one compiler (gcc) for one big-endian machine architecture. Its treatment of bitfields is simple and logical. The results of this program on a modern little-endian machine are very different and very difficult to understand.

Notes on Figure 15.24. Bitfield-structure demonstration.

First box: a bitfield type.

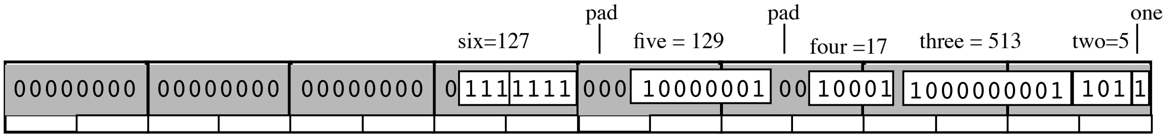
- We declare a bitfield type, `DemoType,`that occupies a minimum of 37 bits. The compiler will add padding, as necessary, to meet the requirements of the local hardware.
- Here are two diagrams of how the data might be laid out in a bitfield structure. Byte boundaries are indicated by divisions in the white strip at the bottom. Short-word boundaries are dark lines that cross the gray area, and field boundaries are indicated by the white boxes in the gray area. This implementation

This program can be used to determine how a compiler and its underlying hardware treat bitfields. The packing used by the Gnu C compiler is diagrammed below.

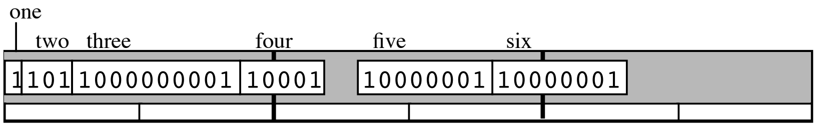
```
#include <iostream>
struct DemoType {
        unsigned int one   : 1;
        unsigned int two   : 3;
        unsigned int three :10;
        unsigned int four  : 5;
        unsigned int       : 2;
        unsigned int five  : 8;
        unsigned int six   : 8;
};

int main( void )
{
    int k;
    unsigned char* bptr;
    DemoType bit = { 1, 5, 513, 17, 129, 0x81 };

    printf( "\n sizeof DemoType = %lu\n", sizeof(DemoType ) );

    printf( "initial values: bit = %u, %u, %u, %u, %u, %u\n",
            bit.one, bit.two, bit.three, bit.four, bit.five, bit.six );
    bptr = (unsigned char*)&bit;
    printf( " hex dump of bit:  %02x %02x %02x %02x %02x %02x %02x %02x\n",
        bptr[7], bptr[6], bptr[5], bptr[4], bptr[3], bptr[2], bptr[1], bptr[0]);

    bit.three = 1023;
    printf( "\n assign 1023 to bit.three: %u, %u, %u, %u, %u, %u\n",
            bit.one, bit.two, bit.three, bit.four, bit.five, bit.six );
    k = bit.two;
    printf( "assign bit.two to k: k = %i\n", k );
    return 0;
}
```

Figure 15.24. Bitfield-structure demonstration.

packs the bits tightly together starting at the left. Fields three and five cross byte boundaries, field four crosses a short-word boundary, and field six crosses a long-word boundary.



- The diagram above shows the way the fields are laid out on a current little-endian machine.
- The diagram below shows the way the fields were laid out on an older big-endian machine. The two layouts are very different! In both cases, the bytes are shown as they would appear in a machine register.



- The standard permits the fields to be assigned memory locations in left-to-right or right-to-left order and padding to be added wherever necessary to meet alignment requirements on the local system.
- Two bits of padding are added between fields four and five because of the unnamed 2-bit field declared there. This is the only instance in the C language where a memory location can be declared without associating a name with it.
- Additional padding bits were placed after field six to fill the structure up to a word boundary, as required by the local compiler.
- One, but not both compilers added padding between fields five and six to avoid having field six cross a long-word boundary.
- The third box prints out the size of the structure on the local system. In the older implementation, it was 6 bytes (48 bits). Therefore, the last 11 bits are padding. Since this system uses *short-word alignment*, all objects occupy an even number of bytes and begin on an even-numbered memory address.
- My current little-endian computer, uses long-word alignment, and the size of the same type is 8 bytes.

Second box: a bitfield object.

- We declare a `DemoType` object named `bit`, and initialize its fields to distinctive values. The syntax for initialization is like the syntax for an ordinary struct, except that no initializer is provided for an unnamed field.
- Each initializer value has the correct number of bits needed to fill its field. All but one of these values starts and ends with a 1 bit and has 0 bits in between. When we look at this in hexadecimal, we can see where each field starts and ends. From this information, we can determine the packing and padding rules used by the local compiler.
- A 2-bit unnamed field lies between fields four and five. An initializer should not supply a value for a padding field; the compiler may store anything there that is convenient. Our compiler initializes these fields to 0 bits.

Third box: The size of the object .

- Please note that C formatted output is being used here. The entire C language is a subset of C++.
- The reason for using C output is simplicity. It is simply easier and briefer to deal with low-level concerns using a low-level language. Here, we want the bytes of the object printed with two columns per byte, separated by spaces. This is easy using `%02x`, and very difficult using `<<`.

Fourth box: making the packing order visible.

In this box, we use a pointer cast and subscripts to look at the contents of each byte of the bitfield object.

- First we print out the contents of each field, simply to verify that the initializer works as expected. This is not always the case. Initializer values too large to fit into the corresponding fields are simply truncated (the leftmost bits are dropped). This is the output from the program in Figure 15.24 when compiled under the `clang` compiler for a Mac running OS 10.11.

```
sizeof DemoType = 8
initial values: bit = 1, 5, 513, 17, 129, 127
hex dump of bit: 00 00 00 7f 10 24 60 1b
```

- We next use a pointer, to enable us to look at the individual bytes of the bitfield object. At the top of `main()`, we declare `bptr` to be `unsigned` because we want to print the values in hexadecimal. We further declare it to be of type `char*` so that we can print individual bytes of the bit vector.
- By using a pointer cast, and storing the result in `bptr`, we are able to use `bptr` with a subscript to access each byte of the bitfield object.
- We cast the address of `bit` to the type of `bptr` and set `bptr` to point at the beginning of `bit`. Then we print out the bytes to verify the position of each field in the whole.
- The `2` in the conversion specifier (`%02x`) causes the values printed to be two columns wide, while the `0` before it means that leading zeros will be printed. This is the form in which the hexadecimal codes are easiest to convert to binary.
- From the first line of output, we know that `bit` occupies 8 bytes in the current implementation. So we use `bptr` to print 8 bytes starting at the address of the beginning of `bit`. Any value of any type can be printed in hex in this way, by using an `unsigned char*`.
- The array is printed backwards (from byte 7 to byte 0) because this code was written and run on a little-endian machine. On a big-endian machine, the subscripts would start at 0 and increase.
- Below we rewrite the hex value from each output field in binary, with spaces separating the bits into groups of four that correspond to the hexadecimal output. We omit the last byte of zeros. The conversion values simply are filled in using the table values in Appendix E, Figure E.4. A `v` is written above the first bit of each of the 6 bytes. Long-word boundaries are marked by `L`, and short-word boundaries by `S`:

  ```
  00        00        00         7f       10        24        60        1b
  L                   S                   L                   S                   L
  v         v         v         v         v         v         v         v         v
  0000 0000 0000 0000 0000 0000 0111 1111 0001 0000 0010 0100 1100 0000 1000 1011
  ```

  We now rewrite the bits using spaces to separate the bitfields. The decimal value of each bitfield is written below it. Padding bits are marked by `o`. We produced the diagram on the prior page from this output.

  ```
  oooooooo oooooooo oooooooo o 1111111 ooo 10000001 oo 1001 10000000001 101 1
                                 127          129     17      513      5 1
  ```

Fifth box: using an individual field. In this box, we use the symbolic part names to look at the logical values in the structure.

- The assignment to `bit.three` demonstrates that an integer value may be assigned to a bitfield in the ordinary way. The C run-time system positions the bits correctly and assigns them to the structure without disturbing the contents of other fields.
- The assignment `k = bit.two` shows that a bitfield may be assigned to an integer in the ordinary way. The C run-time system will lengthen the bitfield to the size of an integer.

  ```
  assign 1023 to bit.three: 1, 5, 1023, 17, 129, 129
  assign bit.two to k: k = 5
  ```

15.5.1 Bitfield Application: A Device Controller (Advanced Topic)

An artist has a studio with a high sloping ceiling containing skylights. Outside, each skylight is covered with louvers that can be opened fully under normal operation to let the light in or closed to protect the glass or keep heat inside the room at night.

The louvers are opened and closed by a small computer-controlled motor, with two limit switches that sense when the skylight is fully open or fully closed. To open the skylight, one runs the motor in the forward direction until the fully open limit switch is activated. To close the skylight, one runs the motor similarly in

Problem scope: Write a program to control a motor that opens and closes a skylight.

Output required: To the screen, a menu that will allow a user to select whether to open the skylight, close it, or report on its current position. To the motor controller, signals to start or stop the motor.

Input: The program receives menu selections from the user. It also receives status codes directly from the motor controller via the memory-mapped address that is its interface to the controller.

Constants: The memory-mapped addresses used by the multifunction chip in this program are `0xffff7100` for the DR and `0xffff7101` for the DDR. Bit positions in the DR, which follow, are numbered starting from the right end of the byte.

Bit #	In or Out?	Purpose	Settings	
0	Output	Motor direction	0 = forward	1 = reverse
1	Output	Motor power	0 = off	1 = on
2	Input	Fully closed louver sensor	0 = not fully closed	1 = fully closed
3	Input	Fully open louver sensor	0 = not fully open	1 = fully open

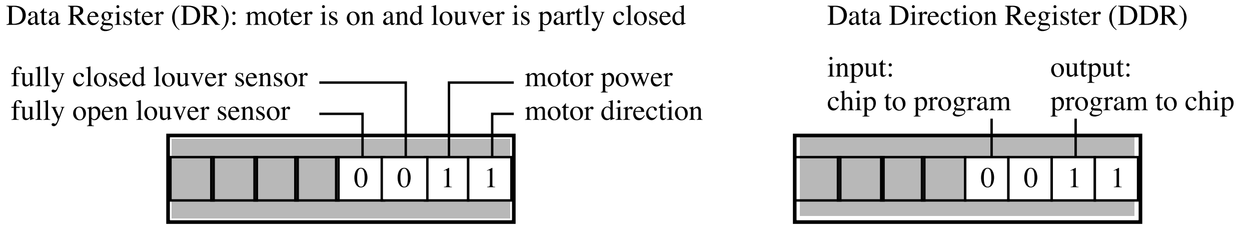

Data Register (DR): moter is on and louver is partly closed
fully closed louver sensor
fully open louver sensor
motor power
motor direction
0 0 1 1
Data Direction Register (DDR)
input:
chip to program
output:
program to chip
0 0 1 1

Figure 15.25. Problem specifications: Skylight controller.

the reverse direction. To know the current location of the skylight, one simply examines the state of the limit switches.

The motor is controlled by a box with relays and other circuitry for selecting its direction, turning it on and off, and sensing the state of the limit switches. The controller box has an interface to the computer through a multifunction chip using a technique known as *memory-mapped I/O*. This means that when certain main memory addresses are referenced, bits are written to or read from the multifunction chip, rather than real, physical memory.

In this program, we assume that the multifunction chip interfaces with the computer through two memory addresses: `0xffff7100` refers to an 8-bit data register (DR) and `0xffff7101` refers to an 8-bit data direction register (DDR). Each bit of the data register can be used to send data either from the hardware to the program, or vice versa. This is controlled by setting the DDR. Data flows from chip to program through a bit of the data register if the corresponding bit of the data direction register is 0. It flows from program to chip if the corresponding DDR bit is 1. The direction of communication, for each bit in the data register, is shown in Figure 15.25.
The program for the skylight controller consists of two parts[7]:

- A set of type declarations (Figure 15.26) and function prototypes.
- A main module containing:
  - A set of constants, initializers, and pointers that connect to the hardware addresses of the two registers.
  - The `main()` function (Figure 15.27) that initializes the system, displays a menu in an infinite loop, and waits for a user to request some service.

[7] The program was debugged using a device simulator, also written in C++ and using threads.

These declarations are stored in the file `skylight.h`, which is used in Figures 15.27 and 15.28.

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

```
// Definitions of the registers on the multifunction chip
struct RegByte {
    unsigned int                 :4;
    unsigned int fullyOpen       :1;
    unsigned int fullyClosed     :1;
    unsigned int motorPower      :1;
    unsigned int motorDirection  :1;
};

typedef RegByte* DevicePointer;
```

```
// Definitions of the codes for the multifunction chip
enum PowerValues      { motorOff = 0, motorOn = 1 };
enum DirectionValues  { motorForward = 0, motorReverse = 1 };
enum Position         { fullyClosed, partOpen, fullyOpen } ;
```

```
// Prototypes for the control operations.
Position skylightStatus( void );
void openSkylight( void );
void closeSkylight( void );
```

Figure 15.26. Declarations for the skylight controller: `skylight.hpp`.

   - Three functions (Figure 15.28) that perform those services: opening the skylight, closing it, and asking about its current state.

Notes on Figure 15.26. Declarations for the skylight controller: `skylight.h`.

First box: the bitfield type.

- We use four bits to communicate with the multifunction chip; two are used by the program to receive status information from the chip and two are used to send control instructions to the chip. As shown in the register diagram, the leftmost four bits in the chip registers will not be used in this application. Therefore, the bitfield type declaration used to model a register begins with an unnamed field for the four padding bits, followed by named fields for two status bits and two control bits.
- We use pointer variables in the program to hold the addresses of the chip registers, so we declare a pointer type that references a register byte.

Second box: status and switch codes.

- Throughout the code, we could deal with 1 and 0 settings, as specified in Figure 15.25. However, doing so makes the code error prone and very hard to understand. Instead, we define three enumerated types to give symbolic names to the various switch settings and status codes. An array of strings is defined in Figure 15.27 parallel to the third enumeration, to allow easy output of the device's status.
- Two of the enumerations are used simply to give names to codes. A series of `#define` commands could be used for this purpose, but `enum` is better because it is shorter and it lets us group the codes into sets of related values. We do not intend to use them as variable or parameter types. It is not necessary to set the symbols equal to 0 and 1 here because those are the default values in an enumeration. However it does make the correspondence between the symbols and the values clear and obvious.

Third box: prototypes. This package is composed of a main program and three functions used to carry out the user's menu commands.

- The functions `openSkylight()` and `closeSkylight` are used to send commands to the device controller.
- The `skylightStatus` function is used to read the status bits set by the controller and determine whether it is an appropriate time to issue a new action command.

Notes on Figure 15.27. A skylight controller.

This program implements the specification of Figure 15.25. It uses the declarations in Figure 15.26 and calls functions in Figure 15.28.

```
#include "skylight.hpp"

// Attach to hardware addresses.
volatile DevicePointer const DR = (DevicePointer)0xffff7100;
volatile DevicePointer const DDR = (DevicePointer)0xffff7101;

// Create initialization constants.
const RegByte DDRmask = {0,0,1,1};  // select output bits
const RegByte DRinit = {0,0,0,0};   // 1 means program -> chip.

// Output strings corresponding to enum constants
const char* enumLabels[] = {"fully closed", "partially open", "fully open"};

int main( void )
{
    char choice;
    const char* menu = " O: Open skylight\n C: Close skylight\n"
                       " R: Report on position\n Q: Quit\n";

    // Initialize chip registers used by application.-----
    *DDR = DDRmask;    // Designate input and output bits.
    *DR = DRinit;      // Make sure motor is turned off.

    for (;;) {
        cout << menu <<"Select operation:  ";
        cin >> choice;
        choice = toupper( choice );
        if (choice == 'Q') break;

        switch (choice) {
            case 'O': openSkylight();                                  break;
            case 'C': closeSkylight();                                 break;
            case 'R': cout <<"Louver is " << enumLabels[ skylightStatus() ] <<"\n";
                                                                       break;
            default:  cout <<"Incorrect choice, try again.\n";
        }
    }
    puts( " Skylight controller terminated.\n" );
    return 0;
}
```

Figure 15.27. A skylight controller.

First box: setting up the registers.

- We want a pointer variable `DR` to point at the address of the data register byte on the multifunction chip. We write its memory-mapped address as a hex literal, cast it to the appropriate pointer type, and store it in `DR`. The keyword `const` after the type name means that `DR` always points at this location and can never be changed.
- The keyword `volatile` means that something outside the program (in this case, the hardware device) may change the value of this variable at unpredictable times. We supply this information to the C++ compiler so that its code optimizer does not eliminate any assignments to or reads from the location that, otherwise, would appear to be redundant.
- Similarly, we set a pointer variable `DDR` to the address of the data direction register byte.
- A bitfield object, DDRmask, is created and will be used by the hardware device when the program is started, to define the communication protocol between device and program. The leftmost two bits are 0 to indicate that they will be used by the program to receive information from the device controller. The last two bits are 1, indicating that the program will use those positions to send control information to the chip.
- Another bitfield object, DRinit, is created for initializing the Data Register to a an off state.

Second box: chip initialization.

- When the system is first turned on, the value `{0,0,1,1}` must be stored in the rightmost bits of the chip's data direction register to indicate the direction of data flow through the bits of the data register. Then the data register can be used to initialize the motor to an off state with a 0 bit vector.
- These two assignments take the bitfield objects that have been prepared and use a single assignment to store them into the two registers of the device controller. Completing the initialization in a single assignment is important when dealing with concurrent devices (computer and controller).

Third box, outer: an infinite loop.

- The remainder of the program is an infinite loop that presents an operation menu to the user and waits for commands. The user can quit at any time (and thereby turn the system off) by choosing option `Q`.
- We display a menu and read a selection. Then we use `toupper()` so that the program recognizes both lower-case and upper-case inputs in the `switch` statement.

Inner box: calling the controller functions.

- The system supports three operations: open the skylight, close it, and report on its position. There is a menu option and a function to call for each. The open and close commands change the state of the skylight using the last two bits of the Data Register. The report function checks on that state in the first two bits of the Data Register and returns a position code that `main()` uses to index the array `enumLabels` and display the position for the user.
- The `switch` has a `default` case to handle input errors. This case is not necessary; without it, an error would just be ignored. However, a good human interface gives feedback after every interaction, and an explicit error comment is helpful.

Notes on Figure 15.28. Operations for the skylight controller.

First box: `skylightStatus()`***.***

- There are three possible states: fully open, fully closed, and somewhere in between (represented by both status bits being off). This information is vital to the program because it must turn off the motor when the skylight reaches either limit.
- The status is tested by checking the appropriate bitfields in `DR` to see which of the mutually exclusive conditions exists and then returning the corresponding position code to `main()`.
- Note that, throughout, we use the symbolic names of the bitfields and the symbolic enum constants to make the code comprehensible.

Second box, outer: `openSkylight()`.

- The basic steps of opening the skylight are turn on the motor, wait until the louvers are completely open, then turn off the motor.
- Sometimes it is not necessary to turn on the motor. If a command is given to open the skylight when it is already open, control simply returns to the menu.
- In between turning on the motor and turning it off, the program sits in a tight `while` loop, waiting for the open status bit in `DR` to change state due to the controller's actions. This technique is referred to as a *busy wait loop*, and is only appropriate for an embedded system.
- Turning the motor off is also done with a single assignment, using a prepared bitfield object.
- Second box, first inner box: turning on the motor.

These functions are called from Figure 15.27 and are in the same source code file.

```
// --- Return skylight position----------------------------------------
position
skylightStatus( void )
{
    if (DR->fullyClosed) return fullyClosed;
    else if (DR->fullyOpen) return fullyOpen;
    else return partOpen;
}
```

```
// --- Open skylight --------------------------------------------------
void
openSkylight( void )
{
    if (DR->fullyOpen) return;  // Don't start motor if already open

    regByte dr = { 0, 0, motorOn, motorForward }; // Prepare start signal.
    *DR = dr;                   // Store start signal in hardware register

    while (!(DR->fullyOpen));   // Wait and watch until louvre is open

    dr.motorPower = motorOff;   // Set control bit to off
    *DR = dr;                   // Store stop signal in hardware register
}
```

```
// --- Close skylight -------------------------------------------------
void
close_skylight( void )
{
    if (DR->fullyClosed) return; // Don't start motor if already closed

    regByte dr = { 0, 0, motorOn, motorReverse };  // Prepare start signal.
    *DR = dr;                    // Store start signal in hardware register
    while (!(DR->fullyClosed));  // Wait and watch until louvre is closed

    dr.motorPower = motorOff;    // Set control bit to off
    *DR = dr;                    // Store stop signal in hardware register
}
```

Figure 15.28. Operations for the skylight controller.

  - If the louver needs to be opened, the program must turn on the motor in the forward direction. In this case, both the direction and power bits of the control byte must be changed.
  - To do this, we prepare a local `RegByte` object with the right control bits.
  - Then the contents of this byte are copied to `DR`, using a single assignment to initiate the motor's action.
- In between turning on the motor and turning it off, the program sits in a tight `while` loop, waiting for the open status bit in `DR` to change state due to the controller's actions[8].
- Second inner box: turning off the motor. A similar pattern is repeated to turn off the motor. The motor power bit in the local `RegByte` is set to off and the byte is transferred to `DR`. The direction bit needs no adjusting since that bit is irrelevant when the motor is off.

***Last box:* `closeSkylight()`.** The `closeSkylight()` function is exactly analogous to `openSkylight()`. It returns promptly if no work is needed. Otherwise, it turns the motor on in the reverse direction, waits for the `fullyClosed` bit to be turned on by the controller, then turns the motor off.

Testing. To run this program in reality, we would need an automated skylight, a motor, a board containing the electronics necessary to drive the motor, and a multifunction chip attached to a computer. However, code always should be tested, if possible, before using it to control a physical device. This program was tested using a skylight simulator: a separate program that emulates the role of the real device, supplies feedback to the program written in this section, and prints periodic reports for the programmer. (This simulation program is too complex to present here in detail.)

Output from a simulation follows. The initial state of the simulated skylight is half open (the result of a power failure). In the output. when you see a line that starts with `SKYLIGHT`, you should imagine that you hear a motor running and see the louvers changing position. To reduce the total amount of output, repetitions of the menu have been replaced by dashed lines.

```
SKYLIGHT: louver is  47% open

Select operation:

         O: Open skylight
         C: Close skylight
         R: Report on position
         Q: Quit

Enter letter of desired item: r

 Louver is partially open
 --------------------------------
 Enter letter of desired item: O
        SKYLIGHT: louver is  60% open
        SKYLIGHT: louver is  80% open
        SKYLIGHT: louver is 100% open
        SKYLIGHT: louver is 101% open
 --------------------------------
 Enter letter of desired item: o
 --------------------------------
 Enter letter of desired item: c
        SKYLIGHT: louver is 100% open
        SKYLIGHT: louver is  80% open
        SKYLIGHT: louver is  60% open
        SKYLIGHT: louver is  40% open
        SKYLIGHT: louver is  20% open
        SKYLIGHT: louver is   0% open
        SKYLIGHT: louver is  -1% open
 --------------------------------
```

[8] This technique is referred to as a busy wait loop, and is only appropriate for a dedicated computer or an embedded system.

Operator	**Meaning and Use**
˜	Used to toggle bit values in vector.
|	Used to combine sections of a vector or turn on bits.
&	Used to isolate sections of a vector or turn off bits.
∼	Used to compare two bit vectors or toggle bits on and off.
<<	Used to shift bits left; fills right end with 0 bits. Shifting an integer left by one bit has the same effect as multiplying it by 2.
>>	Used to shift bits right. Unsigned right shift fills left end with 0 bits; signed right shift fills left end with copies of the sign bit. Shifting an integer right by one bit has the same effect as dividing it by 2.

Figure 15.29. Summary of bitwise operators.

```
Enter letter of desired item: R
Louver is fully closed
--------------------------------
Enter letter of desired item: q
Skylight controller terminated.
```

Note that the motor always "overshoots" by a small amount. This happens because it takes a brief time to turn off the motor after the sensor says that the louver is fully open or fully closed. A real device would have to be calibrated with this in mind.

15.6 What You Should Remember

15.6.1 Major Concepts

Unsigned numbers. At times, applications naturally use numbers whose range always is positive, such as memory addresses and the pixel values of digital images. In these cases, using an `unsigned` type allows for larger ranges and helps prevent errors during calculations.

Hexadecimal. C supports input, output, and literals for hexadecimal integers. Hex notation is important because it translates directly to binary notation and, therefore, lets us work easily with bit patterns.

I/O conversions. We use C formatted output here because it is easier to control the spacing, field length, and leading zeroes in C than it is with C++. All built-in data types have format conversion specifiers. Unsigned data values are displayed using `%u` in decimal form and `%x` in hexadecimal form. For any data type, it is possible to specify a leading fill character other than blank. This is useful for printing leading zeroes in hex values, as in `%08x`.

Bit arrays The bits in an integer data value can be viewed as an array and manipulated using bitmasks and bitwise operators.

Bitmasks. A mask is used to specify which bit or bits within an integer are to be selected for an operation. To create a mask, first write the appropriate bit pattern in binary, then translate it to hexadecimal and `#define` the result as a constant.

Summary of bitwise operators. Bitwise operations are used with bitmasks expressed in hexadecimal notation to operate on both single bits and groups of bits within an integer. Figure 15.29 is a summary of these operators.

Bitfields. Bitfield structures provide a symbolic alternative to the manual manipulation of bits using the bitwise operators. A structure can be declared with components that are smaller than a byte, and those components can be initialized like the fields of an ordinary structure and used like ordinary integers with small ranges. The compiler does all the shifting and masking needed to access the separate bits.

Compression and encryption. Information is compressed to save storage space and transmission time, and/or encrypted to preserve privacy. These applications make extensive use of bitwise operations.

Volatile. When used in a variable declaration, this type qualifier indicates that an outside agent such as a hardware device may change the variable's value at any time. Using the qualifier instructs the compiler not to attempt to optimize the use of the variable.

15.6.2 Programming Style

Constants. As with other constants, use `#define` for bitmasks to generate more understandable code.

Hexadecimal vs. decimal. It is important to *understand* numbers written in hexadecimal, binary, and decimal notations. Since we cannot use binary constants in the code, we need to know when to *use* hexadecimal and when to use decimal notation. Use hexadecimal for literals when working with bitwise operators or interacting with hardware devices because the pattern of bits in the number is important and the translation from hex to binary is direct. Use decimal notation for most other purposes.

Proper data type. In addition to using the appropriate literal form, the proper integer data type should be chosen. The range of needed values should help you to decide between signed and unsigned, as well as the shortest length necessary.

Bit operators vs. bitfields. One must choose whether to use bit operators or bitfield structures as a representation. Bitfield structures are appropriate for describing the bit layout of a hardware device interface that is unlikely to change. They enable us to use field names instead of bit positions to interact with the device more easily. When dealing with a series of bytes from a stream, it is more convenient to use unsigned integers and the bitwise operations, as with the encoding and decoding of information.

15.6.3 Sticky Points and Common Errors

- Some pocket calculators provide a way to enter a number in base 10 and read it out in base 16 or vice versa. If yours does not, use the decimal to binary to hexadecimal method presented in Appendix E to do number conversions. The algorithm that converts directly from base 10 to base 16 relies on modular arithmetic, which is not supported by many calculators.
- When generating bitmasks beware of inverting the bit pattern. Make sure the ones and zeroes are appropriate for the bitwise operation being performed.
- Do not confuse the logical operators `&&`, `||`, and `!` with the bitwise operators `&`, `|`, and `~`. Logical operators do whole-word operations and return only the values `true` or `false`. Bitwise operators use bit vectors of varying length as both input and output.
- Do not confuse the use of `&` and `|` in an expression. Keep straight whether you are using the mask to select bits from a value or to introduce bits into it.
- Beware of shifting a bit vector in the wrong direction, either by using the incorrect operator or mistakenly using a negative shift amount.
- Remember that the data type determines whether a signed bit extension or unsigned zero-fill is performed during right shifts.
- Beware of the relative precedence of the bitwise operators. There are many different precedences, and not all seem natural to someone unfamiliar with manipulating bits.
- Beware of being off by one bit position. It is quite common when shifting or manipulating a particular bit to miss and be off by one position. Knowing the powers of 2 can help reduce the number of times this happens.
- When using bitfields that have lengths shorter than a byte, don't try to store values that are too long to be represented. The most significant bits will be truncated, leading to wrong and confusing results.

15.6.4 New and Revisited Vocabulary

These are the most important terms, concepts, keywords, and operators discussed in this chapter.

binary
decimal
hexadecimal
unsigned integer
hex integer literal
hex character literal
`%x` (hex I/O conversion)
`%u` (unsigned conversion)
two's complement

`~` (complement)
`<<` (left shift)
`>>` (right shift)
signed shift
unsigned shift
`&` (bitwise AND)
`^` (bitwise exclusive OR)
`|` (bitwise OR)
portable notation

bit vector
mask
toggle
unpack and recombine
encode and decode
encryption
bitfield structure
`volatile`
memory-mapped I/O

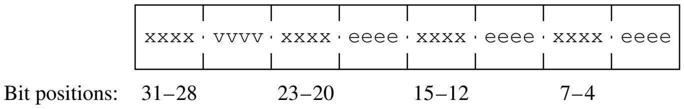
15.7 Exercises

15.7.1 Self-Test Exercises

1. (a) What decimal numbers do you need to convert to get the following hexadecimal outputs: AAA, FAD, F1F1, and B00 ?
   (b) Write the decimal numbers from 15 to 35 in hexadecimal notation.
   (c) Write the decimal value 105 in hexadecimal notation.
   (d) What hexadecimal number equals 101010 in binary?
   (e) Write any string of 16 bits. Show how you convert this to a hexadecimal number. Then write this number as a legal C hexadecimal literal.
2. Compile and run the program in Figure 15.10. Experiment with different input and answer the following questions:

   (a) What happens if you try to print a negative signed integer with a `%hu` format?
   (b) What is the biggest `unsigned int` you can read and write? Note what it looks like in hexadecimal notation; you easily can see why this is the biggest representable integer. Explain it in your own words.
   (c) You can read numbers in hexadecimal format. Try this. Enter a series of numbers and experimentally find one that prints the value 117_{10}.

3. As part of a cryptographic project, 4-byte words are scrambled by moving around groups of 4 bits each. A scrambling pattern follows: it has three sets of bits labeled with the letters `x`, `v`, and `e`. The bits in the `x` positions do not move. All the `e` bits move 8 bits to the left, and the `v` bits go into bits 3. . . 0.



   (a) Define three AND masks to isolate the sets of fields `x`, `v`, and `e`.
   (b) Write three AND instructions to decompose the word by isolating the three sets of bits and storing them in variables named `X`, `V`, and `E`.
   (c) Write the two shift instructions required to reposition the bits of `E` and `V`.
   (d) Write an OR instruction that puts the three parts back together.

4. A college professor is writing a program that emulates the computer he first used in 1960 (an IBM 704). He wants to use this program to teach others about the way in which hardware design has progressed in 40 years. The instructions on this machine were 36 bits long and had four parts:

(a)

Operation code	bits 35–33
Decrement field	bits 32–18
Index register field	bits 17–15
Address field	bits 14–0

	op	decrement	idx	address
bit positions:	35-33	32-18	17-15	14-0

In this machine, the decrement field is used as part of the operation code in some instructions and as a second address field in others. Write a `typedef` and bitfield structure declaration to embed this instruction layout in 6 bytes. Put the necessary padding bits anywhere convenient but make sure that the two 3-bit fields do not cross byte boundaries and the two longer fields cross only one boundary each.

5. Repeat problem 4 using bit masks. Assume that the op and index fields are stored together in one byte and the other fields occupy two bytes each.

6. What is stored in `w` by each line of the following statements? Write out the 8-bit vectors for each variable and show the steps of the operations. Use these values for the variables:

```
unsigned char v, w, x = 1, y = 2, z = 4;
const unsigned char mask = 0xC3;
```

```
a. w = ˜ x & x;                         f. w = ˜ x | x;
b. w = ˜ x ∼ x;                         g. w = 1;   w<<= 1;
c. w = x | y & x | z;                   h. w = x ∼ ˜ y;
d. v = y | z; w = (v<<3) + (v<<1);      i. w = x | y | z>>2;
e. w = x | y & x | z<<y ∼ mask>>x;      j. w = x & y & ˜ z;
```

15.7.2 Using Pencil and Paper

1. (a) Write your full name, including blanks, in the ASCII character code in hexadecimal literal form.
   (b) What base-10 number equals the two's complement binary value 11010110 ?
   (c) Do the following addition in binary two's complement: 01110110 + 10001001 = ?
   (d) Express 00110110_2 in bases 10 and 16.
   (e) Express 5174_{10} in binary and hexadecimal.
   (f) Use binary arithmetic to multiply 00010110_2 by 4_{10}, then express the answer in bases 16 and 10.
   (g) How many times would you need to divide 100110_2 by 2_{10} to get the quotient 1?

2. As part of a cryptographic project, every 4 bytes of information in a data stream will be scrambled. One of the scrambling patterns follows: it has five fields labeled with four letters. The bits in the positions labeled `x` are not moved. The bit field labeled `n` swaps places with the field labeled `v`; the rightmost bit of `n` moves to position 5 and the leftmost bit of `v` shifts to position 27. The field labeled `e` moves 2 bits to the left to make space for this swap.

	xxxx	nnnnnnnnnnn	eee	vvvvvvvvv	xxxxx
Bit positions:	31-28	27-17	16-14	13-5	4-0

   (a) Define four AND masks to isolate the fields `x`, `n`, `e`, and `v`.
   (b) Write four AND instructions to decompose the word, isolating the four sets of bits and storing them in variables named `X`, `N`, `E`, and `V`.
   (c) Write the three shift instructions required to reposition the bits of `N`, `E`, and `V`.
   (d) Write the OR instruction to put the four parts back together.

3. Repeat parts (a) and (b) of exercise 2 using a bitfield structure. Explain why parts (c) and (d) are easier to do using masks than using bitfields.

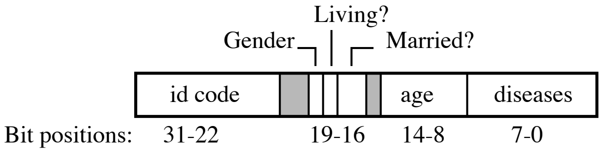
4. (a) Write the numbers from 30_{10} to 35_{10} in binary notation.
   (b) Write 762_{10} in hexadecimal notation.
   (c) Write 762_{10} in binary notation.
   (d) What hexadecimal number equals 11000011_2?
   (e) What base-10 number equals 11000011_2?
   (f) What decimal numbers equal the following hex "words": `C3b0`, `dab`, `add`, and `feed`?

5. (Advanced topic.) The following three riddles have answers related to the limitations of a computer and the quirks of the C language in representing and using the various integer types.

   (a) When is $x + 1 < x$?
   (b) When does a positive number have a negative sign?
   (c) When is a big number small?

6. You are writing a program to analyze health statistics. A data set containing records for a small village has been collected, coded, and stored in a binary file. The data cover about 1,000 residents. Each person's data are packed into a single long unsigned integer, with fields (shown in the diagram) defined as follows:

(a)	ID code	bits 31–22	
	Gender	bit 19	0 = male, 1 = female
	Alive	bit 18	0 = dead, 1 = living
	Marital status	bits 17–16	0 = single, 1 = married, 2 = divorced, 3 = widowed
	Age	bits 14–8	
	Disease code	bits 7–0	



The remaining bit positions (gray in the diagram) are irrelevant to the current research and may be ignored. Write a `typedef` and bitfield structure declaration to describe these data.

7. Solve for `N` in the following equations. Show your work, using an 8-bit unsigned binary representation for the constants and the variables `N` and `T`. Give the answer to each item in binary and hexadecimal format.

   a. $\texttt{N = 21}_{16}\ \texttt{|}\ \texttt{39}_{16}$
   b. $\texttt{3A}_{16}\ \sim\ \texttt{N = 00}_{16}$
   c. $\texttt{N = \~{} AE}_{16}\ \texttt{|}\ \texttt{39}_{16}$
   d. $\texttt{N = 4C}_{16}\ \texttt{>> 1}$
   e. $\texttt{N = 2A}_{16}\ \texttt{\& 0F}_{16}$
   f. $\texttt{N = 4C}_{16}\ \texttt{<< 3}_{16}$
   g. $\texttt{N = A1}_{16}\ \texttt{\& 39}_{16}\ \texttt{| 1E}_{16}$
   h. $\texttt{T = 2A}_{16}\texttt{; \~{}T >> 2}\ \sim\ \texttt{N = 01}_{16}$

15.7.3 Using the Computer

1. The case bit.

   In an alphabetic ASCII character, the third bit from the left is the *case-shift bit*. This bit is 0 for upper-case letters and 1 for lower-case letters. Except for this bit, the code of an upper-case letter is the same as its lower-case counterpart. Write a program that contains a bitmask for the case-shift bit. Read input from the keyboard using `getline( cin, line)`, where `line` is a string variable. Do the operations described below, but make sure to change only alphabetic letters; do not alter the values of numerals, special symbols, and the like. Use the library function `isalpha()` to test the chars.

   Assume that each line, including the last, will end with a newline character.

(a) On the first line of input, use the mask and one bitwise operator to toggle the case of the input; that is, change `B` to `b` or `b` to `B`. Print the results.

(b) On the second line of input, use your mask and one bitwise operator to change the input to all uppercase. Print the results.

(c) On the third line of input, use your mask and one bitwise operator to change the input to all lowercase. Print the results.

2. Decomposing an integer.

   One of the fastest methods of sorting is a *radix sort*. To use this algorithm, each item being sorted is decomposed into fields and the data set is sorted on each field in succession. The speed of the sort depends on the number of fields used; the space used by the sorting array depends on the number of bits in each field. A good compromise for sorting 32-bit numbers is to use four fields of 8 bits each, as in the following diagram. During the sorting process, the data are sorted first based on the least significant field (A, in the diagram) and last using the most significant field (D).

D	C	B	A

   This problem focuses on only the decomposition, not the sorting. Assume you want to sort long integers and need to decompose each into four 8-bit portions, as shown in the diagram. Write a function, `getKey()`, that has two parameters: the number to be decomposed and the pass number, 0. . . 3. Isolate and return the field of the number that is the right key for the specified pass. For pass 0, return field A; for pass 1, field B; and so forth. Write a main program that will let you test your `getKey()` function. For each data value, echo the input in both decimal and hexadecimal form, then call the `getKey()` function four times, printing the four fields in both decimal and hexadecimal format.

3. A computational mystery.

   (a) Enter the following code for the `mystery()` function into a file and write a main program that will call it repeatedly. The main program should enter a pair of numbers, one real and one positive integer; call the `mystery()` function; and print the result.

```
double mystery( double x, int n )/ Assume n >= 0
{
    double y = 1.0;
    while (n > 0) {
        if (n & 1) y *= x;     // Test if n is odd.
        n >>= 1;
        x *= x;                // Square x.
    }
    return y;
}
```

   (b) Manually trace the execution of the code for `mystery()`. Use the argument values `n = 5` and `x = 3.0`, then try again with `x = 2.0` and `n = 6`. On a trace chart, make columns for `x`, `y`, `n`, `n` in binary, and `n & 1`. Show how the values of each expression change every time around the loop.

   (c) Run the program, testing it with several pairs of small numbers. Chart the results. The `mystery()` function uses an unusual method to compute a useful and common mathematical function. Which mathematical function is it?

   (d) Turn in the program, some output, the chart of output results, the hand emulation, and a description of the mathematical function.

4. Binary to decimal conversion.

   Appendix E, Figure E.4 describes and illustrates one method for converting binary numbers to base 10. Write a program to implement this process, as follows:

   (a) Write a function, `strip()`, that returns the value of the rightmost bit of its `long` integer argument. This will be either 0 (`false`) or 1 (`true`). Implement this using bit operations.

(b) Write a void function, convert(), that has one long signed integer argument. It should start by printing a left parenthesis and finish by printing a matching right parenthesis. Between generating these, repeatedly strip a bit off the right end of the argument (or what remains of it). If the bit is a 1, print the corresponding power of 2 and a + sign. Then shift the argument 1 bit to the right. Continue to test, print, and strip until the remaining argument is 0.

(c) Write a main program to test the conversion function, making it easy to test several data values. For each, prompt the user for an integer. If it is negative, print a negative sign and take its absolute value. Then call the convert() function to print the binary value of the number in its expanded form. For example, if the input was -25, the output should be $-(1 + 8 + 16)$.

5. Input hexadecimal, output binary.

Appendix E, Figure E.5 describes and illustrates a method for converting hexadecimal numbers to binary. Write a program to implement this process, as follows:

(a) Define a constant array of char* to implement the bit patterns in the third column of the table in Figure E.5.

(b) Write a function, strip(), that has a call-by-address parameter, np, the number being converted. Use a masking operation to isolate the rightmost 4 bits of np and save this value. Then shift np 4 bits to the right and return this modified value through the parameter. Return the isolated hexadecimal digit as the value of the function.

(c) Write a void function, convert(), that has one long signed integer argument, n, and an empty integer array in which to store an answer. Remember that integers are represented in binary in the computer and 4 binary bits represent the same integer as 1 hexadecimal digit. Repeatedly strip a field of 4 bits off the right end of the argument (or what remains of it) and store the resulting hexadecimal digit in successive slots of the answer array. Do this eight times for a 32-bit integer.

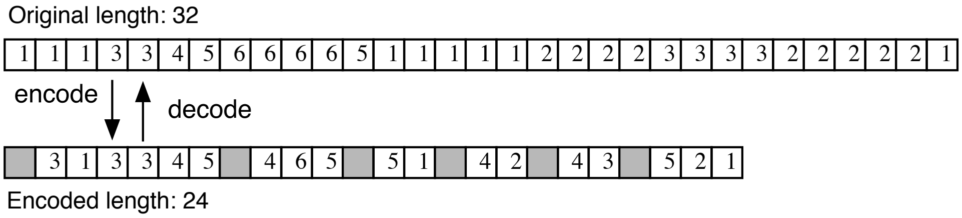
(d) Write a main program to test your conversion function, making it easy to test several data values. For each, prompt the user to enter a hexadecimal integer in the form 10A2, and read it using a %x format specifier. Then call the convert() function, sending in the value of the number and an array to hold the hexadecimal expansion of that number. After conversion, print the number in expanded binary form. Start by printing a left parenthesis, then print the eight hexadecimal digits of the number in their binary form, using the array from part (a). Finish by printing a matching right parenthesis. For example, if the hexadecimal numbers -1 and 17 were entered, the output should be:

```
    ( 1111 1111 1111 1111 1111 1111 1111 1111 )
    ( 0000 0000 0000 0000 0000 0000 0001 0111 )
```

6. Compressing an image—run length encoding.

Digital images often are large and consume vast amounts of disk space. Therefore, it can be desirable to compress them and save space. One simple compression method is *run length encoding.* This technique is based on the fact that most images have large areas of constant intensity. A series of three or more consecutive pixels of the same value is called a *run.* When compressing a file, any value in the input sequence that is not part of a run simply is transferred from the original sequence into the encoded version. When three or more consecutive values are the same, an encoding triplet is created. The first element of this triplet is an escape code, actually a character value not expected to be present in the data. The second value is the number of characters present in the run, stored as a 1-byte unsigned integer. The last value is the input value that is repeated the indicated number of times.

The compression process is illustrated below. A stream of characters is shown first in its original form, and below that in compressed form. The escape code characters are shown as gray boxes.


Original length: 32

1	1	1	3	3	4	5	6	6	6	6	5	1	1	1	1	1	2	2	2	2	3	3	3	3	2	2	2	2	2	1

▨	3	1	3	3	4	5	▨	4	6	5	▨	5	1	▨	4	2	▨	4	3	▨	5	2	1

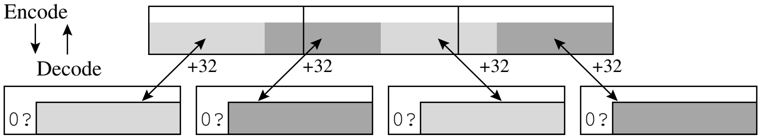
Encoded length: 24

When a compressed image is used, the encoding process must be reversed. All encoding triplets are expanded into a corresponding sequence of identical byte values.

Write a program that will perform run length encoding and decoding. First, open a user-specified file for input (appropriately handle any errors in the file name) and another file for output. Then ask the user whether encoding or decoding should be performed. If encoding, read successive bytes from the input stream, noting any consecutive repetitions. Those occurring once or twice should be echoed to the output file, while runs of length three or more should be converted into an encoding triplet, which then is written to the output file. Use a value of `0xFF` (255) as the escape code for this problem. If a data value of 255 occurs, change it to 254. Any runs in the data longer than 255 should be broken into an appropriate number of triplets of that length, followed by one of lesser length to finish the description. If decoding, read successive bytes from the input stream, scanning for encoding triplets. All characters between these triplets should be copied directly into the output file. Encoding triplets should be expanded to the appropriate length and sent to the output file.

7. Encoding a file—uuencode.

One command in the UNIX operating system is *uuencode*, and its purpose is to encode binary data files in ASCII format for easy transmission between machines. The program will read the contents of one data file and generate another, taking successive groups of three input bytes and transforming them into corresponding groups of four ASCII bytes as follows:



The original 3 bytes (24 bits) are divided into four 6-bit segments. The value of each segment is used to generate a single byte of data. The value of this byte is calculated by adding the value 32 to the 6-bit value and storing the result in a character variable. This creates a printable ASCII value that can be transmitted easily. In the event that the original file length is not divisible by 3, fill in the missing bytes at the end by bytes containing the value 0. The final byte triplet then is converted in the same manner.[9]

Write a program that will perform the uuencode operation. First, open a file with a user-specified name for input (appropriately handle any errors in the file name). Open another file for output. Then read successive byte triplets from the input stream and write corresponding 4-byte chunks to the output stream, adding 0 bytes at the end, if necessary, to make 3 bytes in the final triplet.

8. Decoding a file—uudecode.

The matching command to the uuencode command discussed in the previous exercise is uudecode. It takes as input an ASCII file created using uuencode and produces the corresponding binary file. It simply reverses the encoding process just described, taking 4-byte groups as input and generating 3-byte groups as output. Write a program that will perform the uudecode operation. First, open a file with a user-supplied name for input (appropriately handle any errors in the file name). Open an output file as well. Perform the conversion process by transforming successive 4-byte input groupings into 3-byte output groups and writing them to the output file.

9. Set operations.

A set is a data structure for representing a group of items. The total collection of items that can exist in a set is referred to as the *set's domain.* A set with no items in it is called an *empty set*, while one that contains all the items is called a *universal set.* Items can be added to a set, but if that item already exists in the set, nothing changes. Items also can be removed from the set. In addition, some operations create new sets: (1) *intersection* produces the set of items common to two sets, (2) *union* produces the set containing all items found in either of two sets, (3) *difference* produces the set containing those items in a first set that do not exist in a second set, and (4) *complement* produces the set containing those items in the domain that are not present in a set.

There are many ways to represent a set in C. In this exercise, define a set as a bit vector. Represent the domain of the set as an enumeration. Each bit position in the vector corresponds to an item in the domain. A bit value is 1 if that item is present in the set. Each of the various set operations can be performed by doing an appropriate bitwise operation on sets represented in this manner. Write a program that allows a user to define and manipulate sets in the following manner:

[9] The real uuencode command also generates header and trailer information that is useful in decoding the file. For simplicity, we ignore this information in the exercise. Newer versions add 96 (not 32) to the null character.

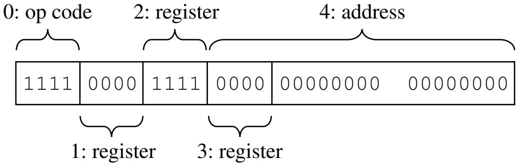
Machine Instruction	**Op Code**	**1st Register**	**2nd Register**	**3rd Register**	**Address**	**Description**
Load	0000	R	—	—	M	Load R with contents of M
Store	0001	R	—	—	M	Store contents of R into M
Add	0010	Ri	Rj	Rk	—	Rk = Ri + Rj
Subtract	0011	Ri	Rj	Rk	—	Rk = Ri - Rj
Multiply	0100	Ri	Rj	Rk	—	Rk = Ri * Rj
Divide	0101	Ri	Rj	Rk	—	Rk = Ri / Rj
Negate	0110	Ri	Rj	—	—	Rj = -Ri
AND	0111	Ri	Rj	Rk	—	Rk = Ri & Rj
OR	1000	Ri	Rj	Rk	—	Rk = Ri | Rj
XOR	1001	Ri	Rj	Rk	—	Rk = Ri ∼ Rj
NOT	1010	Ri	Rj	—	—	Rj = ˜Ri
¡	1011	Ri	Rj	Rk	—	Rk = Ri < Rj
¿	1100	Ri	Rj	Rk	—	Rk = Ri > Rj
=	1101	Ri	Rj	Rk	—	Rk = Ri == Rj
Branch U	1110	—	—	—	M	Branch to address M
Branch C	1111	R	—	—	M	Branch to M if R true

Figure 15.30. Problem 10.

- Let the domain be the colors red, orange, yellow, green, blue, purple, white, and black. To map from a color to a bit in the set representation, start with a bit vector containing the integer value 1 and use the enumeration value as a left shift amount.
- Write a group of functions that will (a) add an element to a set, (b) remove an element from a set, (c) test if an element is in the set, and (d) print the contents of a set. These will be support functions for the rest of the program.
- Write a group of functions that perform set intersection, set union, set difference, and set complement.
- Write a main program that presents an initial menu to the user, the options being the four set operations just mentioned. When one of these options is chosen, the program should ask the user to supply the contents of the necessary sets, perform the operation, and print the contents of the resulting set. To supply the contents of a set, the user should be able to add and remove items from the set repeatedly until the desired group has been produced. When adding or removing an item, the user should be able to type the name of the item (a color) or select it from a menu and the program then should adjust the representation of the set accordingly.

10. Machine code disassembler.

The last phase of the compilation process is turning assembly language into machine language. Sometimes it is desirable to reverse this process by generating assembly language from machine code, or *disassembling.* This is similar to what happens in the instruction decoding step of the CPU. We describe a fictitious machine that has 16 assembly language instructions encoded in a 4-byte machine instruction. This 4-byte area can be decomposed into five pieces. The leftmost half (2 bytes) is split into 4-bit fields. The remaining half may be used in conjunction with the last 4-bit field from the left half to form another large field, as follows:



The contents of these fields depend on the particular instruction, described in Figure 15.30. The operation code determines whether there is a third register number or those bits are to be used as part of a memory address. Write a program that prints the assembly language program described by a machine code file.

First, open a user-specified binary file for input (appropriately handle any errors in the file name). Read successive groups of 4 bytes and treat each group as an instruction. Decode the instruction based on looking at the op code field and interpreting the remaining fields properly. Print an assembly language line describing this. Print register numbers as `R1` or `R15`. Print the 20-bit memory addresses in hexadecimal form. For instance, the two machine instructions on the left might be translated into the assembly instructions on the right:

```
00000101 00001111 11001101 01100100 → LOAD R5, 0xFCD64
00101101 00101011 00000000 00000000 → ADD R13, R2, R11
```