Chapter 5. Arrays, Qualifiers, and Reading Numbers

5.1 Arrays

In constructing our building, we have identified each brick (variable) by name. That process is fine for a small number of bricks, but what happens when we want to construct something larger? We would like to point to a stack of bricks and say, "That's for the left wall. That's brick 1, brick 2, brick 3..."

Arrays allow us to do something similar with variables. An array is a set of consecutive memory locations used to store data. Each item in the array is called an element. The number of elements in an array is called the dimension of the array. A typical array declaration is:

```
/* List of data to be sorted and averaged */
int    data_list[3];
```

The above example declares `data_list` to be an array of three elements. `data_list[0]`, `data_list[1]`, and `data_list[2]` are separate variables. To reference an element of an array, you use a number called the index—the number inside the square brackets ([ ]). C is a funny language that likes to start counting at 0. So, our three elements are numbered to 2.

Common sense tells you that when you declare `data_list` to be three elements long, `data_list[3]` would be valid. Common sense is wrong and `data_list[3]` is illegal.

Example 5-1 computes the total and average of five numbers.

Example 5-1. *array/array.c*

```
[File: array/array.c]
```

```
#include <stdio.h>

float data[5];  /* data to average and total */
float total;    /* the total of the data items */
float average;  /* average of the items */

int main()
{
    data[0] = 34.0;
    data[1] = 27.0;
    data[2] = 45.0;
    data[3] = 82.0;
    data[4] = 22.0;

    total = data[0] + data[1] + data[2] + data[3] + data[4];
    average =  total / 5.0;
    printf("Total %f Average %f\n", total, average);
    return (0);
}
```

This program outputs:

```
Total 210.000000 Average 42.000000
```

5.2 Strings

Strings are sequences of characters. C does not have a built-in string type; instead, strings are created out of character arrays. In fact, strings are just character arrays with a few restrictions. One of these restrictions is that the special character `'\0'` (NUL) is used to indicate the end of a string.

For example:

```
char    name[4];

int main()
{
    name[0] = 'S';
    name[1] = 'a';
    name[2] = 'm';
    name[3] = '\0';
    return (0);
}
```

This code creates a character array of four elements. Note that we had to allocate one character for the end-of-string marker.

String constants consist of text enclosed in double quotes (""). You may have noticed that the first parameter to `printf` is a string constant. C does not allow one array to be assigned to another, so we can't write an assignment of the form:

```
name = "Sam";    /* Illegal */
```

Instead we must use the standard library function `strcpy` to copy the string constant into the variable. (`strcpy` copies the whole string, including the end-of-string character.) To initialize the variable `name` to `Sam`, we would write:

```
#include <string.h>
char   name[4];
int main()
{
    strcpy(name, "Sam");    /* Legal */
    return (0);
}
```

C uses variable-length strings. For example, the declaration:

```
#include <string.h>
char string[50];
int main()
{
    strcpy(string,"Sam");
}
```

creates an array (`string`) that can contain up to 50 characters. The size of the array is 50, but the length of the string is 3. Any string up to 49 characters long can be stored in `string`. (One character is reserved for the NUL that indicates end-of-string.)

String and character constants are very different. Strings are surrounded by double quotes (") and characters by single quotes ('). So "X" is a one-character string, while 'Y' is just a single character. (The string "X" takes up two bytes, one for

the X and one for the end-of-string (\0). The character `'Y'` takes up one byte.)

There are several standard routines that work on string variables, as shown in Table 5-1.

Table 5-1. Partial List of String Functions

Function	Description
`strcpy(`*`string1`*`,` *`string2`*`)`	Copy *`string2`* into *`string1`*
`strcat(`*`string1`*`,` *`string2`*`)`	Concatenate *`string2`* onto the end of *`string1`*
`length` `= strlen(`*`string`*`)`	Get the length of a *`string`*
`strcmp(`*`string1`*`,` *`string2`*`)`	0if *`string1`* equals *`string2`*, otherwise nonzero

The `printf` function uses the conversion `%s` for printing string variables, as shown in Example 5-2.

Example 5-2. *str/str.c*

```
#include <string.h>
#include <stdio.h>
char name[30];    /* First name of someone */
int main()
{
    strcpy(name, "Sam");    /* Initialize the name */
    printf("The name is %s\n", name);
    return (0);
}
```

Example 5-3 takes a first name and a last name and combines the two strings.

The program works by initializing the variable `first` to the first name (Steve). The last name (Oualline) is put in the variable `last`. To construct the full name, the first name is copied into `full_name`. Then `strcat` is used to add a space. We call `strcat` again to tack on the last name.

The dimension of the string variable is 100 because we know that no one we are going to encounter has a name more than 99 characters long. (If we get a name more than 99 characters long, our program will mess up. What actually happens is that you write into memory that you shouldn't access. This access can cause your program to crash, run normally and give incorrect results, or behave in other unexpected ways.)

Example 5-3. *full/full.c*

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

char first[100];       /* first name */
char last[100];        /* last name */
char full_name[200];    /* full version of first and last name */

int main()
{
    strcpy(first, "Steve");      /* Initialize first name */
    strcpy(last, "Oualline");     /* Initialize last name */

    strcpy(full_name, first);     /* full = "Steve" */
    /* Note: strcat not strcpy */
    strcat(full_name, " ");      /* full = "Steve " */
    strcat(full_name, last);      /* full = "Steve Oualline" */

    printf("The full name is %s\n", full_name);
    return (0);
}
```

The output of this program is:

```
The full name is Steve Oualline
```

5.3 Reading Strings

The standard function `fgets` can be used to read a string from the keyboard. The general form of an `fgets` call is:

`fgets(`*`name`*`, sizeof(`*`name`*`), stdin);`

where *name* identifies a string variable. (`fgets` will be explained in detail in Chapter 14.)

The arguments are:

name

> is the name of a character array. The line (including the end-of-line character) is read into this array.

sizeof(name)

> indicates the maximum number of characters to read (plus one for the end-of-string character). The `sizeof` function provides a convenient way of limiting the number of characters read to the maximum numbers that the variable can hold. This function will be discussed in more detail in Chapter 14.

stdin

> is the file to read. In this case, the file is the standard input or keyboard. Other files are discussed in Chapter 14.

Example 5-4 reads a line from the keyboard and reports its length.

Example 5-4. *length/length.c*

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

char line[100]; /* Line we are looking at */

int main()
{
    printf("Enter a line: ");
    fgets(line, sizeof(line), stdin);

    printf("The length of the line is: %d\n", strlen(line));
    return (0);
}
```

When we run this program, we get:

```
Enter a line: test
The length of the line is: 5
```

But the string `test` is only four characters. Where's the extra character coming from? `fgets` includes the end-of-line in the string. So the fifth character is newline (`\n`).

Suppose we wanted to change our `name` program to ask the user for his first and last `name`. Example 5 -5 shows how we could write the program.

Example 5-5. *full1/full1.c*

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

char first[100];       /* First name of person we are working with */
char last[100];        /* His last name */

/* First and last name of the person (computed) */
char full[200];

int main() {
   printf("Enter first name: ");
   fgets(first, sizeof(first), stdin);

   printf("Enter last name: ");
   fgets(last, sizeof(last), stdin);

   strcpy(full, first);
   strcat(full, " ");
   strcat(full, last);

   printf("The name is %s\n", full);
   return (0);
}
```

However, when we run this program we get the results:

```
% name2
Enter first name: John
Enter last name: Doe
The name is John
 Doe
%
```

What we wanted was "John Doe" on the same line. What happened? The `fgets` function gets the entire line, *including the end-of-line*. We must get rid of this character before printing.

For example, the name "John" would be stored as:

```
first[0] = 'J'
first[1] = 'o'
first[2] = 'h'
first[3] = 'n'
first[4] = '\n'
first[5] = '\0'    /* end of string */
```

By setting `first[4]` to NUL (`'\0'`), we can shorten the string by one character and get rid of the unwanted newline. This change can be done with the statement:

```
first[4] = '\0';
```

The problem is that this method will work only for four-character names. We need a general algorithm to solve this problem. The length of this string is the index of the end-of-string null character. The character before it is the one we want to get rid of. So, to trim the string, we use the statement:

```
first[strlen(first)-1] = '\0';
```

Our new program is shown in Example 5-6.

Example 5-6. *full2/full2.c*

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

char first[100];       /* First name of person we are working with */
char last[100];        /* His last name */

/* First and last name of the person (computed) */
char full[200];

int main() {
    printf("Enter first name: ");
    fgets(first, sizeof(first), stdin);
    /* trim off last character */
    first[strlen(first)-1] = '\0';

    printf("Enter last name: ");
    fgets(last, sizeof(last), stdin);
    /* trim off last character */
    last[strlen(last)-1] = '\0';
```

```
    strcpy(full, first);
    strcat(full, " ");
    strcat(full, last);

    printf("The name is %s\n", full);
    return (0);
}
```

Running this program gives us the following results:

```
Enter first name:  John
Enter last name:  Smith
The name is John Smith
```

5.4 Multidimensional Arrays

Arrays can have more than one dimension. The declaration for a two-dimensional array is:

```
type variable[size1][size2];  /* Comment */
```

For example:

```
int matrix[2][4]; /* a typical matrix */
```

Notice that C does *not* follow the notation used in other languages of `matrix[10,12]`.

To access an element of the `matrix`, we use the notation:

```
matrix[1][2] = 10;
```

C allows the programmer to use as many dimensions as needed (limited only by the amount of memory available). Additional dimensions can be tacked on:

```
four_dimensions[10][12][9][5];
```

Question 5-1: *Why does Example 5-7 print the wrong answer?* (Click here for the answer Section 5.15)

Example 5-7. *p_array/p_array.c*

```
#include <stdio.h>

int array[3][2];       /* Array of numbers */

int main()
{
    int x,y;    /* Loop indicies */

    array[0][0] = 0 * 10 + 0;
    array[0][1] = 0 * 10 + 1;
    array[1][0] = 1 * 10 + 0;
    array[1][1] = 1 * 10 + 1;
    array[2][0] = 2 * 10 + 0;
    array[2][1] = 2 * 10 + 1;

    printf("array[%d] ", 0);
    printf("%d ", array[0,0]);
    printf("%d ", array[0,1]);
    printf("\n");

    printf("array[%d] ", 1);
    printf("%d ", array[1,0]);
    printf("%d ", array[1,1]);
    printf("\n");

    printf("array[%d] ", 2);
    printf("%d ", array[2,0]);
    printf("%d ", array[2,1]);
    printf("\n");

    return (0);
}
```

5.5 Reading Numbers

So far, we have only read simple strings, but we want more. We want to read numbers as well. The function `scanf` works like `printf`, except that `scanf` reads numbers instead of writing them. `scanf` provides a simple and easy way of reading numbers *that almost never works* . The function `scanf` is notorious for its poor end-of-line handling, which makes `scanf` useless for all but an expert.

However, we've found a simple way to get around the deficiencies of scanf—we don't use it. Instead, we use fgets to read a line of input and sscanf to convert the text into numbers. (The name sscanf stands for "string scanf". sscanf is like scanf, but works on strings instead of the standard input.)

Normally, we use the variable line for lines read from the keyboard:

```
char line[100];    /* Line of keyboard input */
```

When we want to process input, we use the statements:

```
fgets(line, sizeof(line), stdin);
sscanf(line, format, &variable1, &variable2 . . .;
```

Here fgets reads a line and sscanf processes it. *format* is a string similar to the printf format string. Note the ampersand (&) in front of the variable names. This symbol is used to indicate that sscanf will change the value of the associated variables. (For information on why we need the ampersand, see Chapter 13.)

If you forget to put & in front of each variable for sscanf, the result could be a "Segmentation violation core dumped" or "Illegal memory access" error. In some cases a random variable or instruction will be changed. On UNIX, damage is limited to the current program; however, on MS-DOS/Windows, with its lack of memory protection, this error can easily cause more damage. On MS-DOS/Windows, omitting & can cause a program or system crash.

In Example 5-8, we use sscanf to get and then double a number from the user.

Example 5-8. *double/double.c*

```
[File: double/double.c]
#include <stdio.h>
char  line[100];   /* input line from console */
int   value;      /* a value to double */

int main()
{
```

```
    printf("Enter a value: ");

    fgets(line, sizeof(line), stdin);
    sscanf(line, "%d", &value);

    printf("Twice %d is %d\n", value, value * 2);
    return (0);
}
```

This program reads in a single number and then doubles it. Notice that there is no `\n` at the end of `Enter a value:`. This omission is intentional because we do not want the computer to print a newline after the prompt. For example, a sample run of the program might look like:

```
Enter a value: 12
Twice 12 is 24
```

If we replaced `Enter a value:` with `Enter a value:\n`, the result would be:

```
Enter a value:
12
Twice 12 is 24
```

Question 5-2: *Example 5-9 computes the area of a triangle, given the triangle's width and height. For some strange reason, the compiler refuses to believe that we declared the variable `width`. The declaration is right there on line 2, just after the definition of height. Why isn't the compiler seeing it?* (Click here for the answer Section 5.15)

Example 5-9. *tri/tri.c*

```
#include <stdio.h>
char line[100];/* line of input data */
int  height;   /* the height of the triangle
int  width;    /* the width of the triangle */
int  area;     /* area of the triangle (computed) */

int main()
{
    printf("Enter width height? ");

    fgets(line, sizeof(line), stdin);
    sscanf(line, "%d %d", &width, &height);
```

```
    area = (width * height) / 2;
    printf("The area is %d\n", area);
    return (0);
}
```

5.6 Initializing Variables

C allows variables to be initialized in the declaration statement. For example, the following statement declares the integer counter and initializes it to 0:

```
int counter = 0;    /* number cases counted so far */
```

Arrays can also be initialized in this manner. The element list must be enclosed in curly braces ({}). For example:

```
/* Product numbers for the parts we are making */
int product_codes[3] = {10, 972, 45};
```

The previous initialization is equivalent to:

```
product_codes[0] = 10;
product_codes[1] = 972;
product_codes[2] = 45;
```

The number of elements in {} does not have to match the array size. If too many numbers are present, a warning will be issued. If an insufficient amount of n umbers are present, C will initialize the extra elements to 0.

If no dimension is given, C will determine the dimension from the number of elements in the initialization list. For example, we could have initialized our variable product_codes with the state ment:

```
/* Product numbers for the parts we are making */
int product_codes[] = {10, 972, 45};
```

Initializing multidimensional arrays is similar to initializing single-dimension arrays. A set of brackets ([ ]) encloses each dimension. The declaration:

```
int matrix[2][4]; /* a typical matrix */
```

can be thought of as a declaration of an array of dimension 2 with elements that are arrays of dimension 4. This array is initialized as follows:

```
/* a typical matrix */
```

```
int matrix[2][4] =
    {
        {1, 2, 3, 4},
        {10, 20, 30, 40}
    };
```

Strings can be initialized in a similar manner. For example, to initialize the variable name to the string "Sam", we use the statement:

```
char    name[] = {'S', 'a', 'm', '\0'};
```

C has a special shorthand for initializing strings: Surround the string with double quotes ("") to simplify initialization. The previous example could have been written:

```
char name[] = "Sam";
```

The dimension of name is 4, because C allocates a place for the '\0' character that ends the string.

The following declaration:

```
char string[50] = "Sam";
```

is equivalent to:

```
char string[50];
.
.
.
strcpy(string,"Sam");
```

An array of 50 characters is allocated but the length of the string is 3.

5.7 Types of Integers

C is considered a medium-level language because it allows you to get very close to the actual hardware of the machine. Some languages, like BASIC [1] , go to great lengths to completely isolate the user fro m the details of how the processor works. This simplification comes at a great loss of efficiency. C lets you give detailed information about how the hardware is to be used.

[1] Some more advanced versions of BASIC do have number types. However, for this example, we are talking about basic BASIC.

For example, most machines let you use different length numbers. BASIC provides the programmer with only one numeric type. Though this restriction simplifies the programming, BASIC programs are extremely inefficient. C allows the programmer to specify many different flavors of integers, so that the programmer can make best use of hardware.

The type specifier **int** tells C to use the most efficient size (for the machine you are using) for the integer. This can be two to four bytes depending on the machine. (Some less common machines use strange integer sizes such as 9 or 40 bits.)

Sometimes you need extra digits to store numbers larger than those allowed in a normal **int**. The declaration:

```
long int answer;    /* the result of our calculations */
```

is used to allocate a long integer. The **long** qualifier informs C that we wish to allocate extra storage for the integer. If we are going to use small numbers and wish to reduce storage, we use the qualifier **short**. For example:

```
short int year;          /* Year including the 19xx part */
```

C guarantees that the size of storage for **short** < = **int** < = **long**. In actual practice, **short** almost always allocates two bytes, **long** four bytes, and **int** two or four bytes. (See Appendix B, for numeric ranges.)

The type **short int** usually uses 2 bytes, or 16 bits. 15 bits are used normally for the number and 1 bit for the sign. This format gives the type a range of -32768 (-2^{15}) to 32767 ($2^{15} - 1$). An **unsigned short int** uses all 16 bits for the number, giving it the range of to 65535 (2^{16}). All **int** declarations default to **signed**, so that the declaration:

```
signed long int answer;    /* final result */
```

is the same as:

```
long int answer;          /* final result */
```

Finally, we consider the very short integer of type **char**. Character variables use 1 byte. They can also be used for numbers in the range of -128 to 127 (**signed char**) or to 255 (**unsigned char**). Unlike integers, they do not default to **signed**; the default is compiler dependent.[2] Very short integers may be printed using the integer conversion (%d).

[2] Turbo C++ and GNU's gcc even have a command-line switch to make the default for type **char** either **signed** or **unsigned**.

You cannot read a very short integer directly. You must read the number into an integer and then use an assignment statement. For example:

```
#include <stdio.h>
signed char ver_short;   /* A very short integer */
char line[100];         /* Input buffer */
int temp;              /* A temporary number */

int main()
{
    /* Read a very short integer */
    fgets(line, sizeof(line), stdin);
    sscanf(line, "%d", &temp);
    very_short = temp;
}
```

Table 5 -2 contains the `printf` and `sscanf` conversions for integers.

Table 5-2. Integer printf/sscanf Conversions

%Conversion	Uses
%hd	(signed) short int
%d	(signed) int
%ld	(signed) long int
%hu	unsigned short int
%u	unsigned int
%lu	unsigned long int

The range of the various flavors of integers is listed in Appendix B.

long int declarations allow the program to explicitly specify extra precision where it is needed (at the expense of memory). **short int** numbers save space but have a more limited range. The most compact integers have type **char**. They also have the most limited range.

unsigned numbers provide a way of doubling the positive range at the expense of eliminating negative numbers. They are also useful for things that can never be negative, like counters and indices.

The flavor of number you use will depend on your program and storage requirements.

5.8 Types of Floats

The **float** type also comes in various flavors. **float** denotes normal precision (usually 4 bytes). **double** indicates double precision (usually 8 bytes). Double-precision variables give the programmer many times the range and precision of single-precision (**float**) variables.

The qualifier **long double** denotes extended precision. On some systems, this is the same as **double**; on others, it offers additional precision. All types of floating-point numbers are always signed.

Table 5-3 contains the `printf` and `sscanf` conversions for floating-point numbers.

Table 5-3. Float printf/sscanf Conversions

% Conversion	Uses	Notes
`%f`	`float`	`printf` only.[3]
`%lf`	`double`	`scanf` only.
`%Lf`	`long double`	Not available on all compilers.

[3] The %f format works for printing double and float because of an automatic conversion built into C's parameter passing.

On some machines, single-precision, floating-point instructions execute faster (but less accurately) than double-precision instructions. Double-precision instructions gain accuracy at the expense of time and storage. In most cases, **float** is adequate; however, if accuracy is a problem, switch to **double**. (See Chapter 16.)

5.9 Constant Declarations

Sometimes you want to use a value that does not change, such as . The keyword **const** indicates a variable that never changes. For example, to declare a value for `PI`, we use the statement:

```
const float PI = 3.1415927;    /* The classic circle constant */
```

By convention, variable names use only lowercase and constant names use only uppercase. However, the language does not require this case structure,

and some exotic coding styles use a different convention.

Constants must be initialized at declaration time and can never be changed. For example, if we tried to reset the value of PI to 3.0, we would generate an error message:

```
PI = 3.0;      /* Illegal */
```

Integer constants can be used as a size parameter when declaring an array:

```
/* Max. number of elements in the total list.*/
const int TOTAL_MAX = 50;
float total_list[TOTAL_MAX]; /* Total values for each category */
```

This way of specifying the use of integer constants is a relatively new innovation in the C language and is not yet fully supported by all compilers.

5.10 Hexadecimal and Octal Constants

Integer numbers are specified as a string of digits, such as 1234, 88, -123, etc. These strings are decimal (base 10) numbers: 174 or 17410. Computers deal with binary (base 2) numbers: 10101110. The octal (base 8) system easily converts to and from binary. Each group of three digits ($2^3 = 8$) can be transformed into a single octal digit. Thus, 10101110 can be written as 10 101 110 and changed to the octal 256. Hexadecimal (base 16) numbers have a similar conversion; only 4 bits are used at a time.

The C language has conventions for representing octal and hexadecimal values. Leading zeros are used to signal an octal constant. For example, 0123 is 123 (octal) or 83 (decimal). Starting a number with "0x" indicates a hexadecimal (base 16) constant. So, 0x15 is 21 (decimal). Table 5-4 shows several numbers in all three bases.

Table 5-4. Integer Examples

Base 10	Base 8	Base 16
6	06	0x6

9	011	0x9
15	017	0xF

5.11 Operators for Performing Shortcuts

C not only provides you with a rich set of declarations, but also gives you a large number of special-purpose operators.

Frequently, the programmer wants to increment (increase by 1) a variable. Using a normal assignment statement, this operation would look like:

```
total_entries = total_entries + 1;
```

C provides us with a shorthand for performing this common task. The++ operator is used for incrementing:

```
++total_entries;
```

A similar operator, `--`, can be used for decrementing (decreasing by 1) a variable:

```
--number_left;
/* is the same as */
number_left = number_left - 1;
```

But suppose that we want to add 2 instead of 1. Then we can use the following notation:

```
total_entries += 2;
```

This notation is equivalent to:

```
total_entries = total_entries + 2;
```

Each of the simple operators, as shown in Table 5-5, can be used in this manner.

Table 5-5. Shorthand Operators

Operator	Shorthand	Equivalent Statement
+=	`x += 2;`	`x = x + 2;`
-=	`x -= 2;`	`x = x - 2;`

*=	x *= 2;	x = x * 2;
/=	x /= 2;	x = x / 2;
%=	x %= 2;	x = x % 2;

5.12 Side Effects

Unfortunately, C allows the programmer to use *side effects*. A side effect is an operation that is performed in addition to the main operation executed by the statement. For example, the following is legal C code:

```
size = 5;
result = ++size;
```

The first statement assigns to `size` the value of 5. The second statement assigns to `result` the value of `size` (main operation) and increments `size` (side effect).

But in what order are these processes performed? There are four possible answers.

1. `result` is assigned the value of `size` (5), and then `size` is incremented. `result` is 5 and `size` is 6.
2. `size` is incremented, and then `result` is assigned the value of `size` (6). `result` is 6 and `size` is 6.
3. The answer is compiler-dependent and varies from computer to computer.
4. If we don't write code like this, then we don't have to worry about such questions.

The correct answer is number 2: the increment occurs before the assignment. However, number 4 is a much better answer. The main effects of C are confusing enough without having to worry about side effects.

Some programmers value compact code very highly. This attitude is a holdover from the early days of computing when storage cost a significant amount of money. I believe that the art of programming has evolved to the point where clarity is much more valuable than compactness. (Great novels, which a lot of people enjoy reading, are *not* written in shorthand.)

C actually provides two flavors of the ++ operator. One is *variable*++ and the other is ++*variable*. The first:

```
number = 5;
result = number++;
```

evaluates the expressions, and then increments the number; `result` is 5. The second:

```
number = 5;
result = ++number;
```

increments the number first, and then evaluates the expression; `result` is 6. However, using `++` or `--` in this way can lead to some surprising code:

```
o = --o - o--;
```

The problem with this line is that it looks as if someone wrote Morse code. The programmer doesn't read this statement, but rather decodes it. If we never use `++` or `--` as part of any other statement, and instead always put them on lines by themselves, the difference between the two flavors of these operators will not be noticeable.

5.13 ++x or x++

The two forms of the increment operator are called the prefix form (++x) and the postfix form (x++). Which form should you use? Actually in C your choice doesn't matter. However, if you use C++ with its overloadable operators, the prefix version (++x) is more efficient.[4] So, to develop good habits for learning C++, use the prefix form.[5]

[4] For details, see the book *Practical C++ Programming* (O'Reilly & Associates).

[5] Consider the irony of a language with its name in postfix form (C++) working more efficiently with prefix forms of the increment and decrement operators. Maybe the name should be ++C.

5.14 More Side-Effect Problems

More complex side effects can confuse even the C compiler. Consider the following code fragment:

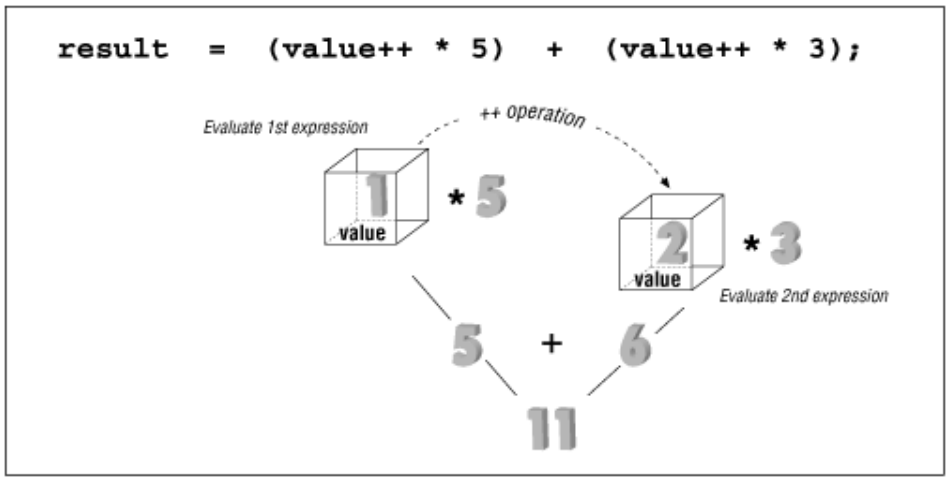
```
value = 1;
result = (value++ * 5) + (value++ * 3);
```

This expression tells C to perform the following steps:

1. Multiply value by 5, and add 1 to value.
2. Multiply value by 3, and add 1 to value.
3. Add the results of the two multiplications together.

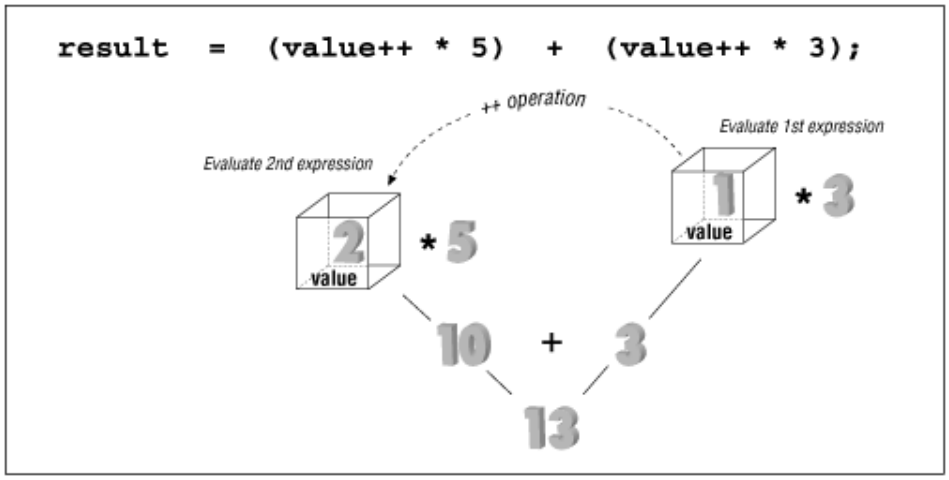
Steps 1 and 2 are of equal priority (unlike in the previous example) so the compiler can choose the order of execution. Suppose the compiler executes step 1 first, as shown in Figure 5-1.

Figure 5-1. Expression evaluation method 1



Or suppose the compiler executes step 2 first, as shown in Figure 5-2.

Figure 5-2. Expression evaluation method 2



By using the first method, we get a result of 11. By using the second method, we get a result of 13. The result of this expression is ambiguous. Depending on how the compiler was implemented, the result may be 11 or 13. Even worse, some compilers change the behavior if optimization is turned on. So what was "working" code may break when optimized.

By using the operator ++ in the middle of a larger expression, we created a problem. (This problem is not the only problem that ++ and -- can cause. We will get into more trouble in Chapter 10.)

In order to avoid trouble and keep the program simple, always put ++ and -- on a line by themselves.

5.15 Answers

Answer 5-1: The problem is the use of the expression `array[x,y]` in the `printf` statement:

```
printf("%d ", array[x,y]);
```

Each index to a multidimension array must be placed inside its own set of square brackets ([ ]). The statement should read:

```
printf("%d ", array[x][y]);
```

For those of you who want to read ahead a little, the comma operator can be used to string multiple expressions together. The value of this operator is the value of the last expressions. As a result `x,y` is equivalent to `y;` and `array[y]` is actually a pointer to row y of the array. Because pointers have strange values, the `printf` outputs strange results. (See Chapter 17, and Chapter 21.)

Answer 5-2: The programmer accidentally omitted the end comment (`*/`) after the comment for height. The comment continues onto the next line and engulfs the declaration, as shown in Example 5-10.

Example 5-10. Comment Answer

```
#include <stdio.h>
char line[100];/* line of input data */
int  height;   /* the height of the triangle
int  width;    /* the width of the triangle */
int  area;     /* area of the triangle (computed) */

int main()
{
    printf("Enter width height? ");

    fgets(line, sizeof(line), stdin);
    sscanf(line, "%d %d", &width, &height);

    area = (width * height) / 2;
    printf("The area is %d\n", area);
    return (0);
}
```

Consider another minor problem with this program. If `width` and `height` are both odd, we get an answer that's slightly wrong. (How would you correct this error?)

5.16 Programming Exercises

Exercise 5-1: Write a program that converts Centigrade to Fahrenheit.

$$F = \frac{9}{5}C + 32$$

Exercise 5-2: Write a program to calculate the volume of a sphere.

$$\frac{4}{3}\pi r^3$$

Exercise 5-3: Write a program that prints the perimeter of a rectangle given its height and width. *perimeter* = *2* ·(*width* + *height*)

Exercise 5-4: Write a program that converts kilometers per hour to miles per hour. *miles* = (*kilometer* ·*0.6213712*

Exercise 5-5: Write a program that takes hours and minutes as input, and then outputs the total number of minutes. (1 hour 30 minutes = 90 minutes).

Exercise 5-6: Write a program that takes an integer as the number of minutes, and outputs the total hours and minutes (90 minutes = 1 hour 30 minutes).