CHAPTER 1

The C Language

This chapter focuses on some basic elements of the C programming language. When you begin programming in C, you probably will find yourself coming up with basic questions regarding the conventions, keywords, and terms used with the C language. This chapter attempts to answer some of the most frequently asked questions regarding these subjects.

For instance, one of the most commonly used constructs of the C language is the `switch` statement. This chapter includes three frequently asked questions regarding this powerful language element. This chapter also covers several other topics such as loops, branching, operator precedence, and blocking guidelines. When reading this chapter, pay close attention to the questions regarding the `switch` statement and operator precedence, because these elements of the C language sometimes can be confusing for the beginning C programmer.

I.1: What is a local block?

Answer:

A local block is any portion of a C program that is enclosed by the left brace ({) and the right brace (}). A C function contains left and right braces, and therefore anything between the two braces is contained in a local block. An `if` statement or a `switch` statement can also contain braces, so the portion of code between these two braces would be considered a local block. Additionally, you might want to create your own local block

without the aid of a C function or keyword construct. This is perfectly legal. Variables can be declared within local blocks, but they must be declared only at the *beginning* of a local block. Variables declared in this manner are visible only within the local block. Duplicate variable names declared within a local block take precedence over variables with the same name declared outside the local block. Here is an example of a program that uses local blocks:

```
#include <stdio.h>

void main(void);

void main()
{

     /* Begin local block for function main() */

     int test_var = 10;

     printf("Test variable before the if statement: %d\n", test_var);

     if (test_var > 5)
     {

          /* Begin local block for "if" statement */

          int test_var = 5;

          printf("Test variable within the if statement: %d\n",
                 test_var);

          {

               /* Begin independent local block (not tied to
                  any function or keyword) */

               int test_var = 0;

               printf(
               "Test variable within the independent local block:%d\n",
               test_var);

          }

          /* End independent local block */

     }

     /* End local block for "if" statement */

     printf("Test variable after the if statement: %d\n", test_var);

}

/* End local block for function main() */
```

This example program produces the following output:

```
Test variable before the if statement: 10
Test variable within the if statement: 5
```

```
Test variable within the independent local block: 0
Test variable after the if statement: 10
```

Notice that as each `test_var` was defined, it took precedence over the previously defined `test_var`. Also notice that when the `if` statement local block had ended, the program had reentered the scope of the original `test_var`, and its value was 10.

Cross Reference:

I.2: Should variables be stored in local blocks?

I.2: Should variables be stored in local blocks?

Answer:

The use of local blocks for storing variables is unusual and therefore should be avoided, with only rare exceptions. One of these exceptions would be for debugging purposes, when you might want to declare a local instance of a global variable to test within your function. You also might want to use a local block when you want to make your program more readable in the current context. Sometimes having the variable declared closer to where it is used makes your program more readable. However, well-written programs usually do not have to resort to declaring variables in this manner, and you should avoid using local blocks.

Cross Reference:

I.1: What is a local block?

I.3: When is a *switch* statement better than multiple *if* statements?

Answer:

A `switch` statement is generally best to use when you have more than two conditional expressions based on a single variable of *numeric* type. For instance, rather than the code

```
if (x == 1)
     printf("x is equal to one.\n");
else if (x == 2)
     printf("x is equal to two.\n");
else if (x == 3)
     printf("x is equal to three.\n");
else
     printf("x is not equal to one, two, or three.\n");
```

the following code is easier to read and maintain:

```
switch (x)
{
     case 1:   printf("x is equal to one.\n");
```

```
                    break;
     case 2:   printf("x is equal to two.\n");
                    break;
     case 3:   printf("x is equal to three.\n");
                    break;
     default:  printf("x is not equal to one, two, or three.\n");
                    break;
}
```

Notice that for this method to work, the conditional expression must be based on a variable of *numeric* type in order to use the `switch` statement. Also, the conditional expression must be based on a single variable. For instance, even though the following `if` statement contains more than two conditions, it is not a candidate for using a `switch` statement because it is based on string comparisons and not numeric comparisons:

```
char* name = "Lupto";

if (!stricmp(name, "Isaac"))
     printf("Your name means 'Laughter'.\n");
else if (!stricmp(name, "Amy"))
     printf("Your name means 'Beloved'.\n ");
else if (!stricmp(name, "Lloyd"))
     printf("Your name means 'Mysterious'.\n ");
else
     printf("I haven't a clue as to what your name means.\n");
```

Cross Reference:

I.4: Is a default case necessary in a `switch` statement?

I.5: Can the last case of a `switch` statement skip including the `break`?

I.4: Is a default case necessary in a *switch* statement?

Answer:

No, but it is not a bad idea to put `default` statements in `switch` statements for error- or logic-checking purposes. For instance, the following `switch` statement is perfectly normal:

```
switch (char_code)
{
     case 'Y':
     case 'y': printf("You answered YES!\n");
               break;
     case 'N':
     case 'n': printf("You answered NO!\n");
               break;
}
```

Consider, however, what would happen if an unknown character code were passed to this `switch` statement. The program would not print anything. It would be a good idea, therefore, to insert a default case where this condition would be taken care of:

```
...
     default:  printf("Unknown response: %d\n", char_code);
               break;
...
```

Additionally, default cases come in handy for logic checking. For instance, if your `switch` statement handled a fixed number of conditions and you considered any value *outside* those conditions to be a logic error, you could insert a default case which would flag that condition. Consider the following example:

```
void move_cursor(int direction)
{
     switch (direction)
     {
          case UP:     cursor_up();
                       break;
          case DOWN:   cursor_down();
                       break;
          case LEFT:   cursor_left();
                       break;
          case RIGHT:  cursor_right();
                       break;
          default:     printf("Logic error on line number %ld!!!\n",
                              __LINE__);
                       break;
     }
}
```

Cross Reference:

I.3: When is a `switch` statement better than multiple `if` statements?

I.5: Can the last case of a `switch` statement skip including the `break`?

I.5: Can the last case of a *switch* statement skip including the *break*?

Answer:

Even though the last case of a `switch` statement does not require a `break` statement at the end, you should add `break` statements to all cases of the `switch` statement, including the last case. You should do so primarily because your program has a strong chance of being maintained by someone other than you who might add cases but neglect to notice that the last case has no `break` statement. This oversight would cause what would formerly be the last case statement to "fall through" to the new statements added to the bottom of the `switch` statement. Putting a `break` after each case statement would prevent this possible mishap and make your program more "bulletproof." Besides, most of today's optimizing compilers will *optimize out* the last break, so there will be no performance degradation if you add it.

Cross Reference:

I.3: When is a `switch` statement better than multiple `if` statements?

I.4: Is a default case necessary in a `switch` statement?

I.6: Other than in a *for* statement, when is the comma operator used?

Answer:

The comma operator is commonly used to separate variable declarations, function arguments, and expressions, as well as the elements of a `for` statement. Look closely at the following program, which shows some of the many ways a comma can be used:

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

void main(void);

void main()
{

     /* Here, the comma operator is used to separate
        three variable declarations. */

     int i, j, k;

     /* Notice how you can use the comma operator to perform
        multiple initializations on the same line. */

     i = 0, j = 1, k = 2;

     printf("i = %d, j = %d, k = %d\n", i, j, k);

     /* Here, the comma operator is used to execute three expressions
        in one line: assign k to i, increment j, and increment k.
        The value that i receives is always the rightmost expression. */

     i = (j++, k++);

     printf("i = %d, j = %d, k = %d\n", i, j, k);

     /* Here, the while statement uses the comma operator to
        assign the value of i as well as test it. */

     while (i = (rand() % 100), i != 50)
          printf("i is %d, trying again...\n", i);

     printf("\nGuess what? i is 50!\n");

}
```

Notice the line that reads

```
i = (j++, k++);
```

This line actually performs three actions at once. These are the three actions, in order:

1. Assigns the value of k to i. This happens because the left value (lvalue) always evaluates to the rightmost argument. In this case, it evaluates to k. Notice that it does not evaluate to k++, because k++ is a postfix incremental expression, and k is not incremented until the assignment of k to i is made. If the expression had read ++k, the value of ++k would be assigned to i because it is a prefix incremental expression, and it is incremented before the assignment is made.
2. Increments j.
3. Increments k.

Also, notice the strange-looking `while` statement:

```
while (i = (rand() % 100), i != 50)
     printf("i is %d, trying again...\n");
```

Here, the comma operator separates two expressions, each of which is evaluated for each iteration of the `while` statement. The first expression, to the left of the comma, assigns `i` to a random number from 0 to 99. The second expression, which is more commonly found in a `while` statement, is a conditional expression that tests to see whether `i` is not equal to 50. For each iteration of the `while` statement, `i` is assigned a new random number, and the value of `i` is checked to see that it is not 50. Eventually, `i` is randomly assigned the value 50, and the `while` statement terminates.

Cross Reference:

I.12: Is left-to-right or right-to-left order guaranteed for operator precedence?

I.13: What is the difference between `++var` and `var++`?

I.7: How can you tell whether a loop ended prematurely?

Answer:

Generally, loops are dependent on one or more variables. Your program can check those variables outside the loop to ensure that the loop executed properly. For instance, consider the following example:

```
#define REQUESTED_BLOCKS 512

int x;
char* cp[REQUESTED_BLOCKS];

/* Attempt (in vain, I must add...) to
   allocate 512 10KB blocks in memory. */


for (x=0; x< REQUESTED_BLOCKS; x++)
{

     cp[x] = (char*) malloc(10000, 1);

     if (cp[x] == (char*) NULL)
          break;

}
```

```
/* If x is less than REQUESTED_BLOCKS,
   the loop has ended prematurely. */

if (x < REQUESTED_BLOCKS)
     printf("Bummer! My loop ended prematurely!\n");
```

Notice that for the loop to execute successfully, it would have had to iterate through 512 times. Immediately following the loop, this condition is tested to see whether the loop ended prematurely. If the variable `x` is anything less than 512, some error has occurred.

Cross Reference:

None.

I.8: What is the difference between *goto* and *longjmp()* and *setjmp()*?

Answer:

A `goto` statement implements a *local* jump of program execution, and the `longjmp()` and `setjmp()` functions implement a *nonlocal*, or far, jump of program execution. Generally, a jump in execution of any kind should be avoided because it is not considered good programming practice to use such statements as `goto` and `longjmp` in your program.

A `goto` statement simply bypasses code in your program and jumps to a predefined position. To use the `goto` statement, you give it a labeled position to jump to. This predefined position must be within the same function. You cannot implement `goto`s between functions. Here is an example of a `goto` statement:

```
void bad_programmers_function(void)
{

     int x;

     printf("Excuse me while I count to 5000...\n");

     x = 1;

     while (1)
     {
          printf("%d\n", x);

          if (x == 5000)
               goto all_done;
          else
               x = x + 1;
     }

all_done:

     printf("Whew! That wasn't so bad, was it?\n");

}
```

This example could have been written much better, avoiding the use of a `goto` statement. Here is an example of an improved implementation:

```
void better_function(void)
{

     int x;

     printf("Excuse me while I count to 5000...\n");

     for (x=1; x<=5000; x++)
          printf("%d\n", x);

     printf("Whew! That wasn't so bad, was it?\n");

}
```

As previously mentioned, the `longjmp()` and `setjmp()` functions implement a nonlocal `goto`. When your program calls `setjmp()`, the current state of your program is saved in a structure of type `jmp_buf`. Later, your program can call the `longjmp()` function to restore the program's state as it was when you called `setjmp()`. Unlike the `goto` statement, the `longjmp()` and `setjmp()` functions do not need to be implemented in the same function. However, there is a major drawback to using these functions: your program, when restored to its previously saved state, will lose its references to any dynamically allocated memory between the `longjmp()` and the `setjmp()`. This means you will waste memory for every `malloc()` or `calloc()` you have implemented between your `longjmp()` and `setjmp()`, and your program will be horribly inefficient. It is highly recommended that you avoid using functions such as `longjmp()` and `setjmp()` because they, like the `goto` statement, are quite often an indication of poor programming practice.

Here is an example of the `longjmp()` and `setjmp()` functions:

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

jmp_buf saved_state;

void main(void);
void call_longjmp(void);

void main(void)
{

     int ret_code;

     printf("The current state of the program is being saved...\n");

     ret_code = setjmp(saved_state);

     if (ret_code == 1)
     {
          printf("The longjmp function has been called.\n");
          printf("The program's previous state has been restored.\n");
          exit(0);
     }
```

```
    printf("I am about to call longjmp and\n");
    printf("return to the previous program state...\n");

    call_longjmp();

}

void call_longjmp(void)
{

    longjmp(saved_state, 1);

}
```

Cross Reference:

None.

I.9: What is an lvalue?

Answer:

An lvalue is an expression to which a value can be assigned. The lvalue expression is located on the *left side* of an assignment statement, whereas an rvalue (see FAQ I.11) is located on the *right side* of an assignment statement. Each assignment statement must have an lvalue and an rvalue. The lvalue expression must reference a storable variable in memory. It cannot be a constant. For instance, the following lines show a few examples of lvalues:

```
int x;
int* p_int;

x = 1;
*p_int = 5;
```

The variable `x` is an integer, which is a storable location in memory. Therefore, the statement `x = 1` qualifies `x` to be an lvalue. Notice the second assignment statement, `*p_int = 5`. By using the `*` modifier to reference the area of memory that `p_int` points to, `*p_int` is qualified as an lvalue. In contrast, here are a few examples of what would not be considered lvalues:

```
#define CONST_VAL 10

int x;

/* example 1 */
1 = x;

/* example 2 */
CONST_VAL = 5;
```

In both statements, the left side of the statement evaluates to a constant value that cannot be changed because constants do not represent storable locations in memory. Therefore, these two assignment statements *do not* contain lvalues and will be flagged by your compiler as errors.

Cross Reference:

I.10: Can an array be an lvalue?

I.11: What is an rvalue?

I.10: Can an array be an lvalue?

Answer:

In FAQ I.9, an lvalue was defined as an expression to which a value can be assigned. Is an array an expression to which we can assign a value? The answer to this question is no, because an array is composed of several separate *array elements* that cannot be treated as a whole for assignment purposes. The following statement is therefore illegal:

```
int x[5], y[5];

x = y;
```

You could, however, use a `for` loop to iterate through each element of the array and assign values individually, such as in this example:

```
int i;
int x[5];
int y[5];

...
for (i=0; i<5; i++)
     x[i] = y[i]
...
```

Additionally, you might want to copy the whole array all at once. You can do so using a library function such as the `memcpy()` function, which is shown here:

```
memcpy(x, y, sizeof(y));
```

It should be noted here that unlike arrays, structures *can* be treated as lvalues. Thus, you can assign one structure variable to another structure variable of the same type, such as this:

```
typedef struct t_name
{
     char last_name[25];
     char first_name[15];
     char middle_init[2];
} NAME;

...

NAME my_name, your_name;

...

your_name = my_name;

...
```

In the preceding example, the entire contents of the `my_name` structure were copied into the `your_name` structure. This is essentially the same as the following line:

```
memcpy(your_name, my_name, sizeof(your_name));
```

Cross Reference:

I.9: What is an lvalue?

I.11: What is an rvalue?

I.11: What is an rvalue?

Answer:

In FAQ I.9, an lvalue was defined as an expression to which a value can be assigned. It was also explained that an lvalue appears on the *left side* of an assignment statement. Therefore, an rvalue can be defined as an expression that can be assigned to an lvalue. The rvalue appears on the *right side* of an assignment statement. Unlike an lvalue, an rvalue can be a constant or an expression, as shown here:

```
int x, y;

x = 1;              /* 1 is an rvalue; x is an lvalue */

y = (x + 1);        /* (x + 1) is an rvalue; y is an lvalue */
```

As stated in FAQ I.9, an assignment statement must have both an lvalue and an rvalue. Therefore, the following statement would not compile because it is missing an rvalue:

```
int x;

x = void_function_call()  /* the function void_function_call()
                             returns nothing */
```

If the function had returned an integer, it would be considered an rvalue because it evaluates into something that the lvalue, `x`, can store.

Cross Reference:

I.9: What is an lvalue?

I.10: Can an array be an lvalue?

I.12: Is left-to-right or right-to-left order guaranteed for operator precedence?

Answer:

The simple answer to this question is neither. The C language does not always evaluate left-to-right or right-to-left. Generally, function calls are evaluated first, followed by complex expressions and then simple

expressions. Additionally, most of today's popular C compilers often rearrange the order in which the expression is evaluated in order to get better optimized code. You therefore should *always* implicitly define your operator precedence by using parentheses.

For example, consider the following expression:

```
a = b + c/d / function_call() * 5
```

The way this expression is to be evaluated is totally ambiguous, and you probably will not get the results you want. Instead, try writing it by using implicit operator precedence:

```
a = b + (((c/d) / function_call()) * 5)
```

Using this method, you can be assured that your expression will be evaluated properly and that the compiler will not rearrange operators for optimization purposes.

Cross Reference:

None.

I.13: What is the difference between *++var* and *var++*?

Answer:

The ++ operator is called the increment operator. When the operator is placed *before* the variable (`++var`), the variable is incremented by 1 before it is used in the expression. When the operator is placed *after* the variable (`var++`), the expression is evaluated, and then the variable is incremented by 1. The same holds true for the decrement operator (`--`). When the operator is placed before the variable, you are said to have a prefix operation. When the operator is placed after the variable, you are said to have a postfix operation.

For instance, consider the following example of postfix incrementation:

```
int x, y;

x = 1;
y = (x++ * 5);
```

In this example, postfix incrementation is used, and `x` is not incremented until after the evaluation of the expression is done. Therefore, `y` evaluates to 1 times 5, or 5. After the evaluation, `x` is incremented to 2.

Now look at an example using prefix incrementation:

```
int x, y;

x = 1;
y = (++x * 5);
```

This example is the same as the first one, except that this example uses prefix incrementation rather than postfix. Therefore, `x` is incremented before the expression is evaluated, making it 2. Hence, `y` evaluates to 2 times 5, or 10.

Cross Reference:

None.

I.14: What does the modulus operator do?

Answer:

The modulus operator (`%`) gives the *remainder* of two divided numbers. For instance, consider the following portion of code:

```
x = 15/7
```

If `x` were an integer, the resulting value of `x` would be 2. However, consider what would happen if you were to apply the modulus operator to the same equation:

```
x = 15%7
```

The result of this expression would be the remainder of 15 divided by 7, or 1. This is to say that 15 divided by 7 is 2 with a remainder of 1.

The modulus operator is commonly used to determine whether one number is evenly divisible into another. For instance, if you wanted to print every third letter of the alphabet, you would use the following code:

```
int x;

for (x=1; x<=26; x++)
     if ((x%3) == 0)
          printf("%c", x+64);
```

The preceding example would output the string `"cfilorux"`, which represents every third letter in the alphabet.

Cross Reference:

None.