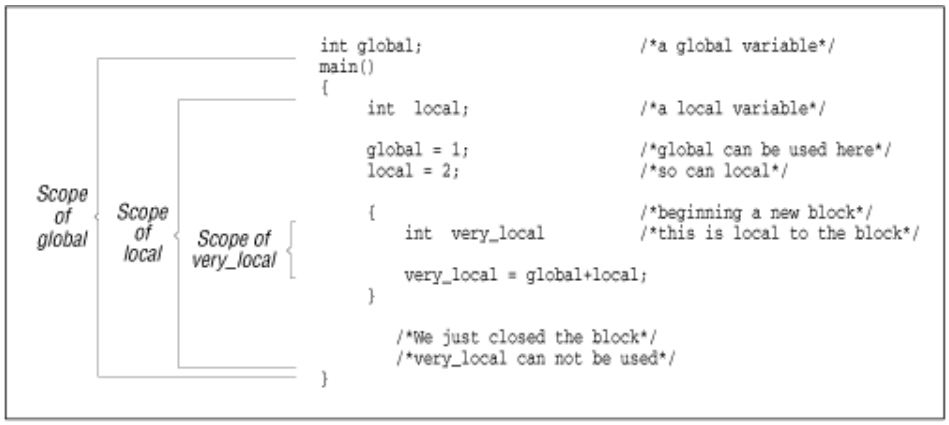
Chapter 9. Variable Scope and Functions

So far, we have been using only global variables. In this chapter, we will learn about other kinds of variables and how to use them. This chapter also tells you how to divide your code into functions.

9.1 Scope and Class

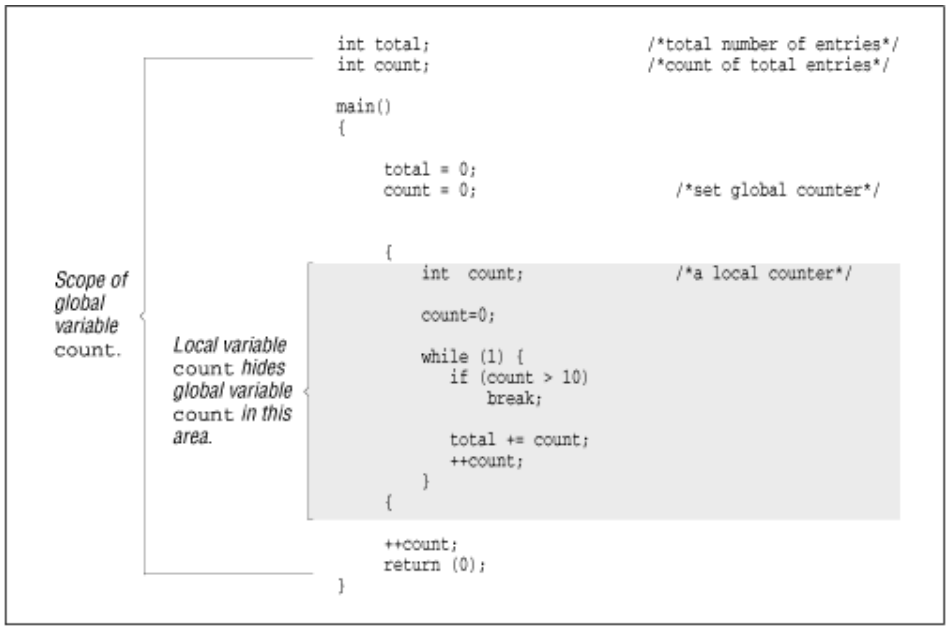
All variables have two attributes: scope and class. The scope of a variable is the area of the program in which the variable is valid. A *global variable* is valid everywhere (hence the name global), so its scope is the whole program. A *local variable* has a scope that is limited to the *block* in which it is declared and cannot be accessed outside that block. A *block* is a section of code enclosed in curly braces ({}). Figure 9-1 shows the difference between local and global variables.

Figure 9-1. Local and global variables



You can declare a local variable with the same name as a global variable. Normally, the scope of the variable count (first declaration) would be the whole program. The declaration of a second local count takes precedence over the global declaration inside the small block in which the local count is declared. In this block, the global count is *hidden* . You can also nest local declarations and hide local variables. Figure 9-2 illustrates a hidden variable.

Figure 9-2. Hidden variables



The variable count is declared as both a local variable and a global variable. Normally, the scope of count (global) is the entire program; however, when a variable is declared inside a block, that instance of the variable becomes the active one for the length of the block. The global count has been hidden by the local count for the scope of this block. The shaded block in the figure shows where the scope of count (global) is hidden.

A problem exists in that when you have the statement:

```
count = 1;
```

you cannot tell easily to which count you are referring. Is it the global count, the one declared at the top of main, or the one in the middle of the **while** loop? You should give these variables different names, like `total_count`, `current_count`, and `item_count`.

The *class* of a variable may be either *permanent* or *temporary*. Global variables are always permanent. They are created and initialized before the program starts and remain until it terminates. Temporary variables are allocated from a section of memory called the *stack* at the beginning of the block. If you try to allocate too many temporary variables, you will get a "Stack overflow" error. The space used by the temporary variables is returned to the stack at the end of the block. Each time the block is entered, the temporary variables are initialized.

The size of the stack depends on the system and compiler you are using. On many UNIX systems, the program is automatically allocated the largest possible stack. On other systems, a default stack size is allocated that can be changed by a compiler switch. On MS-DOS/Windows systems, the stack space must be less than 65,536 bytes. This may seem like a lot of space; however, several large arrays can eat it up quickly. You should consider making all large arrays permanent.

Local variables are temporary unless they are declared **static**.

static has an entirely different meaning when used with global variables. It indicates that a variable is local to the current file. See Chapter 18.

Example 9 -1 illustrates the difference between permanent and temporary variables. We have chosen obvious names: `temporary` is a temporary variable, while `permanent` is permanent. The variable `temporary` is initialized each time it is created (at the beginning of the **for** statement block). The variable `permanent` is initialized only once, at startup time.

In the loop, both variables are incremented. However, at the top of the loop, `temporary` is initialized to one, as shown in Example 9-1.

Example 9-1. *vars/vars.c*

```
#include <stdio.h>

int main() {
    int counter;    /* loop counter */
    for (counter = 0; counter < 3; ++counter) {
        int temporary = 1;       /* A temporary variable */
        static int permanent = 1; /* A permanent variable */


        printf("Temporary %d Permanent %d\n",
           temporary, permanent);
        ++temporary;
        ++permanent;
    }
    return (0);
}
```

The output of this program is:

```
Temporary 1 Permanent 1
Temporary 1 Permanent 2
Temporary 1 Permanent 3
```

Temporary variables are sometimes referred to as *automatic* variables because the space for them is allocated automatically. The qualifier **auto** can be used to denote a temporary variable; however, in practice it is almost never used.

Table 9 -1 describes the different ways in which a variable can be declared.

Table 9-1. Declaration Modifiers

Declared	Scope	Class	Initialized
Outside all blocks	Global	Permanent	Once
static outside all blocks	Global[1]	Permanent	Once
Inside a block	Local	Temporary	Each time block is entered
static inside a block	Local	Permanent	Once

[1] A **static** declaration made outside blocks indicates the variable is local to the file in which it is declared. (See Chapter 18 for more information on programming with multiple files.)

9.2 Functions

Functions allow us to group commonly used code into a compact unit that can be used repeatedly. We have already encountered one function, `main`. It is a special function called at the beginning of the program. All other functions are directly or indirectly called from `main`.

Suppose we want to write a program to compute the area of three triangles. We could write out the formula three times, or we could create a function to do the work. Each function should begin with a comment block containing the following:

Name

Name of the function

Description

Description of what the function does

Parameters

Description of each of the parameters to the function

Returns

Description of the return value of the function

Additional sections may be added such as file formats, references, or notes. Refer to Chapter 3 , for other suggestions.

Our function to compute the area of a triangle begins with:

```
/*****************************************
 * triangle -- Computes area of a triangle.  *
 *                                       *
 * Parameters                              *
 *   width -- Width of the triangle.         *
 *   height -- Height of the triangle.       *
 *                                       *
 * Returns                                 *
 *   area of the triangle.                   *
 *****************************************/
```

The function proper begins with the line:

```
float triangle(float width, float height)
```

float is the function type. The two parameters are `width` and `height`. They are of type **float** also.

C uses a form of parameter passing called "Call by value". When our procedure `triangle` is called, with code such as:

```
triangle(1.3, 8.3);
```

C copies the value of the parameters (in this case 1.3 and 8.3) into the function's parameters (`width` and `height`) and then starts executing the function's code. With this form of parameter passing, a function cannot pass data back to the caller using parameters.[2]

[2] This statement is not strictly true. We can trick C into passing information back through the use of pointers, as we'll see in Chapter 13.

The function type is not required by C. If no function type is declared, the type defaults to **int**. However, if no type is provided, the maintainer cannot determine if you wanted to use the default (**int**) or if you simply forgot to declare a type. To avoid this confusion, always declare the function type and do not use the default.

The function computes the area with the statement:

```
area = width * height / 2.0;
```

What's left is to give the result to the caller. This step is done with the **return** statement:

```
return (area);
```

Example 9-2 shows our full triangle function.

Example 9-2. *tri-sub/tri-sub.c*

```
#include <stdio.h>
/*****************************************
 * triangle -- Computes area of a triangle. *
 *                                       *
 * Parameters                              *
 *   width -- Width of the triangle.       *
 *   height -- Height of the triangle.      *
 *                                       *
 * Returns                                *
 *   area of the triangle.                  *
 *****************************************/
float triangle(float width, float height)
{
    float area;     /* Area of the triangle */

    area = width * height / 2.0;
    return (area);
}
```

The line:

```
size = triangle(1.3, 8.3);
```

is a call to the function `triangle`. C assigns `1.3` to the parameter `width` and `8.3` to `height`.

If functions are the rooms of our building, then parameters are the doors between the rooms. In this case, the value `1.3` is going through the door marked `width`. Parameters' doors are one way. Things can go in, but they can't go out. The **return** statement is how we get data out of the function. In our `triangle` example, the function assigns the local variable `area` the value `5.4`, then executes the statement `return (area);`.

The return value of this function is `5.4`, so our statement:

```
size = triangle (1.3, 8.3)
```

assigns `size` the value `5.4`.

Example 9-3 computes the area of three triangles.

Example 9-3. *tri-prog/tri-prog.c*

```
[File: tri-sub/tri-prog.c]

#include <stdio.h>

/*****************************************
 * triangle -- Computes area of a triangle. *
 *                                       *
 * Parameters                              *
 *   width -- Width of the triangle.       *
 *   height -- Height of the triangle.      *
 *                                       *
 * Returns                                *
 *   area of the triangle.                  *
 *****************************************/
float triangle(float width, float height)
{
    float area;     /* Area of the triangle */

    area = width * height / 2.0;
    return (area);
}

int main()
{
```

```
    printf("Triangle #1 %f\n", triangle(1.3, 8.3));
    printf("Triangle #2 %f\n", triangle(4.8, 9.8));
    printf("Triangle #3 %f\n", triangle(1.2, 2.0));
    return (0);
}
```

If we want to use a function before we define it, we must declare it just like a variable to inform the compiler about the function. We use the declaration:

```
/* Compute a triangle */
float triangle (float width, float height);
```

for the `triangle` function. This declaration is called the *function prototype*.

The variable names are not required when declaring a function prototype. Our prototype could have just as easily been written as:

```
float triangle(float, float);
```

However, we use the longer version because it gives the programmer additional information, and it's easy to create prototypes using the editor's cut and paste functions.

Strictly speaking, the prototypes are optional for some functions. If no prototype is specified, the C compiler assumes the function returns an **int** and takes any number of parameters. Omitting a prototype robs the C compiler of valuable information that it can use to check function calls. Most compilers have a compile-time switch that warns the programmer about function calls without prototypes.

9.3 Functions with No Parameters

A function can have any number of parameters, including none. But even when using a function with no parameters, you still need the parentheses:

```
value = next_index();
```

Declaring a prototype for a function without parameters is a little tricky. You can't use the statement:

```
int next_index();
```

because the C compiler will see the empty parentheses and assume that this is a K&R-style function declaration. See Chapter 19, for details on this older style. The keyword **void** is used to indicate an empty parameter list. So the prototype for our `next_index` function is:

```
int next_index(void);
```

void is also used to indicate that a function does not return a value. (Void is similar to the FORTRAN subroutine or PASCAL procedure.) For example, this function just prints a result; it does not return a value:

```
void print_answer(int answer)
{
    if (answer < 0) {
       printf("Answer corrupt\n");
       return;
    }
    printf("The answer is %d\n", answer);
}
```

Question 9-1: *Example 9-4 should compute the length of a string.*[3] *Instead, it insists that all strings are of length 0. Why?* (Click here for the answer Section 9.6)

[3] This function performs the same function as the library function `strlen`.

Example 9-4. *len/len.c*

```
/*******************************************************
 * Question:                                           *
 *      Why does this program always report the length  *
 *     of any string as 0?                               *
 *                                                     *
 * A sample "main" has been provided.  It will ask      *
 * for a string and then print the length.             *
 *******************************************************/
#include <stdio.h>

/*******************************************************
 * length -- Computes the length of a string.          *
 *                                                     *
 * Parameters                                           *
 *      string -- The string whose length we want.      *
 *                                                     *
 * Returns                                              *
 *      the length of the string.                       *
 *******************************************************/
int  length(char string[])
{
    int           index;     /* index into the string */
```

```
    /*
     * Loop until we reach the end of string character
     */
    for (index = 0; string[index] != '\0'; ++index)
       /* do nothing */
    return (index);
}

int main()
{
     char line[100];    /* Input line from user */

     while (1) {
       printf("Enter line:");
       fgets(line, sizeof(line), stdin);

       printf("Length (including newline) is: %d\n", length(line));
    }
}
```

9.4 Structured Programming

Computer scientists spend a great deal of time and effort studying how to program. The result is that they come up with absolutely, positively, the best programming methodology—a new one each month. Some of these systems include flow charts, top-down programming, bottom-up programming, structured programming, and object-oriented design (OOD).

Now that we have learned about functions, we can talk about using *structured programming techniques* to design programs. These techniques are ways of dividing up or structuring a program into small, well-defined functions. They make the program easy to write and easy to understand. I don't claim that this method is the absolute best way to program. It happens to be the method that works best for me. If another system works better for you, use it.

The first step in programming is to decide what you are going to do. This has already been described in Chapter 7. Next, decide how you are going to structure your data.

Finally, the coding phase begins. When writing a paper, you start with an outline of each section in the paper described by a single sentence. The details will be filled in later. Writing a program is a similar process. You start with an outline, and this becomes your main function. The details can be hidden within other functions. For example, Example 9-5 solves all the world's problems.

Example 9-5. Solve the World's Problems

```
int main()
{
    init();
    solve_problems();
    finish_up();
    return (0);
}
```

Of course, some of the details will have to be filled in later.

Start by writing the main function. It should be less than three pages long. If it grows longer, consider splitting it up into two smaller, simpler functions. After the main function is complete, you can start on the others.

This type of structured programming is called *top-down programming*. You start at the top (`main`) and work your way down.

Another type of coding is called *bottom-up programming*. This method involves writing the lowest-level function first, testing it, and then building on that working set. I tend to use some bottom-up techniques when I'm working with a new standard function that I haven't used before. I write a small function to make sure that I really know how the function works, and then continue from there. This approach is used in Chapter 7 to construct the calculator program.

So, in actual practice, both techniques are useful. A mostly top-down, partially bottom-up technique results. Computer scientists have a term for this methodology: chaos. The one rule you should follow in programming is "Use what works best."

9.5 Recursion

Recursion occurs when a function calls itself directly or indirectly. Some programming functions, such as the factorial, lend themselves naturally to recursive algorithms.

A recursive function must follow two basic rules:

- It must have an ending point.
- It must make the problem simpler.

A definition of factorial is:

```
fact(0) = 1
```

```
fact(n) = n * fact(n-1)
```

In C, this definition is:

```
int fact(int number)
{
    if (number == 0)
       return (1);
    /* else */
    return (number * fact(number-1));
}
```

This definition satisfies our two rules. First, it has a definite ending point (when `number == 0`). Second, it simplifies the problem because the calculation of `fact(number-1)` is simpler than `fact(number)`.

Factorial is legal only for `number >= 0`. But what happens if we try to compute `fact(-3)`? The program will abort with a stack overflow or similar message. `fact(-3)` calls `fact(-4)`, which calls `fact(-5)`, etc. No ending point exists. This error is referred to as an infinite recursion error.

Many things that we do iteratively can be done recursively—for example, summing up the elements of an array. We define a function to add elements *m*-*n* of an array as follows:

- If we have only one element, then the sum is simple.
- Otherwise, we use the sum of the first element and the sum of the rest.

In C, this function is:

```
int sum(int first, int last, int array[])
{
    if (first == last)
       return (array[first]);
    /* else */
    return (array[first] + sum(first+1, last, array));
}
```

For example:

```
Sum(1 8 3 2) =
    1 + Sum(8 3 2) =
        8 + Sum(3 2) =
            3 + Sum (2) =
```

```
             2
          3 + 2 = 5
       8 + 5 = 13
   1 + 13 = 14
Answer = 14
```

9.6 Answers

Answer 9 -1: The programmer went to a lot of trouble to explain that the **for** loop did nothing (except increment the index). However, there is no semicolon (;) at the end of the **for**. C keeps on reading until it sees a statement (in this case `return(index)`), and then puts that statement in the **for** loop. Properly done, this program should look like Example 9-6.

Example 9-6. *len2/len2.c*

```
#include <stdio.h>

int  length(char string[])
{
    int           index;     /* index into the string */

    /*
     * Loop until we reach the end-of-string character
     */
    for (index = 0; string[index] != '\0'; ++index)
       continue; /* do nothing */
    return (index);
}

int main()
{
     char line[100];    /* Input line from user */

     while (1) {
       printf("Enter line:");
       fgets(line, sizeof(line), stdin);

       printf("Length (including newline) is: %d\n", length(line));
    }
}
```

9.7 Programming Exercises

Exercise 9-1: Write a procedure that counts the number of words in a string. (Your documentation should describe exactly how you define a word.) Write a program to test your new procedure.

Exercise 9-2: Write a function `begins(string1,string2)` that returns true if `string1` begins `string2`. Write a program to test the function.

Exercise 9-3: Write a function `count(number, array, length)` that counts the number of times `number` appears in `array`. The array has `length` elements. The function should be recursive. Write a test program to go with the function.

Exercise 9-4: Write a function that takes a character array and returns a primitive hash code by adding up the value of each character in the array.

Exercise 9-5: Write a function that returns the maximum value of an array of numbers.

Exercise 9-6: Write a function that scans a character array for the character `-` and replaces it with `_`.