Chapter 7
Functions

7.1 Chapter Overview

Functions are logical groupings of code, a series of steps, that are given a name. Functions are especially useful when these series of steps will need to be done over and over again. `printf()` and `scanf()` are functions that we have already been using. They exist in the C library because they are common routines that are used all the time; this way, we don't have to re-write the code for printing to the screen or receiving inputs. These functions hide the complexity of performing I/O operations and provide a simple abstraction for us to use.

In addition to using pre-defined functions, we can also write our own. Functions that we write ourselves exist as part of the program and are used the same way, by calling it by it's name that we've given it and using it the way we have defined it.

7.2 Defining Functions

We have already been writing our own functions – the `main()` part of all our programs is a function. Recall Program 2.1 from the book, page 11:

```
1 int main(void) {
2   printf ("Programming is fun. \n");
3   return 0;
4 }
```

While not especially interesting (all this program does is prints out a statement to the display screen), it does provide a full example of the definition of a function. Let's examine this function line-by-line.

There are several points of interest:

1. First, on line 1, the line starts out with `int`. This is the ***return type*** of the function being defined. This specifies what, if anything, the function *returns* to the caller. In this example, the `int` return type says that this function will return an integer to the caller. Functions use return values to communicate information from the function to the caller.
2. The next point of interest on line 1 is the word `main`. This is the name we would use to *call* on the function to do some work for us. The function called "`main`" is special – it defines the entry point of the application.
3. Next, we see `(void)`. This is the ***parameter list*** of the function. Functions use ***parameters*** to communicate information from the caller to the function. In this case, "`void`" indicates that the caller will not pass any information to the function.
4. Finally, we see a `{`. This open curly brace indicates the beginning of the program block that defines the *body* of the function (which must eventually be closed). Between the curly braces is the *body* of the function. This specifies a sequence of statements that will be executed when the function is called. Because the function is declared to return `int`, the function must have a `return` statement that returns an integer value. That return statement is found on line 3. Finally, the definition of the function ends with the close curly brace on line 4.

Now consider a more advanced example:

```
1 int square(int x) {
2   return x * x;
3 }
```

This function is capable of calculating the square of an integer. The square of an integer is an integer, so the return type of the function is `int`. The function's name is `square`. The function takes a single parameter, `x` (called a *formal parameter*) and that parameter's type is `int`. This means that when the function is called, the caller must supply the function with a single integer parameter.

7.3 Calling Functions

Now that we have a useful function, let's consider how we would use it. Before we can call a function, we must know what that function's ***signature*** is. The *signature* of a function includes the function's name, the return type, the number of arguments that function takes (if any), and the types of those arguments. This signature forms a ***function prototype*** that is placed before the `main` function. Again, this is not new (although it has been hidden). The file `stdio.h` includes prototypes for the functions we've been using all along (e.g., `printf()`, `scanf()`).

Now let's consider a full example that uses the `square()` function described on the previous page:

```
1  #include <stdio.h>      // Include prototypes of printf and scanf
2
3  int square(int x);      // Prototype for the 'square' function
4
5  int main(void) {
6    int value = 2;
7    int valueSquared = square(value);
8
9    printf("%d\n", square(4));
10
11   return 0;
12 }
13
14 int square(int x) {
15   return x * x;
16 }
```

This example begins by including `stdio.h`. As mentioned previously, this file includes prototypes for the standard I/O functions we commonly use. Next, we include prototypes of the functions defined within this file (with the exception of `main`). After that, we define the `main` function just as we always have. The only difference is, now we have a new function at our disposal, `square`.

On line 7, the function `square` is *called*, passing to it, the variable `value`. Because `square` takes a single `int` parameter, and because `value` is an integer, the call is correct. Also, because `square` *returns* an `int`, we can use that value, which is *assigned* to another variable, namely `valueSquared`.

Similarly, on line 9, the value `4` is passed to the `square` function, and the return value is passed as a parameter to `printf` to satisfy the `%d` place-holder. Again, `square` expects a single integer parameter, and `4` is an integer literal; `square` returns an integer, and `printf`'s `%d` place-holder expects an integer – everything is correct.

7.4 Parameter Passing

In the previous example, you'll notice that both an integer variable and an integer literal can be passed to `square`. In fact, any integer expression can be used. You could, for example, call `square` like:

```
square(1 + 1);
square(square(1));
square(square(1) + 1);
square(square(1) + square(1));
```

Any expression that evaluates to an `int` can be passed to a function expecting an `int`. When passing parameters (other than arrays – more on this later), the values are automatically *copied* into the formal parameter and then changes are made only to that *local* variable. Functions cannot change the value of the argument that was passed to it – only it's local copy of that variable.

The `printf` function that we are used to using can take in 1 or more arguments. An example of each follows:

```
printf("Programming is fun!\n");
printf("%d\n", sum);
```

7.5 Automatic Local Variables

Variables defined inside a function are known as **automatic local variables**, or just **local variables,** because they are automatically created ***each time the function is called,*** and ***they are known* only *locally within that function***. The values of local variables can be accessed only by the function in which they are defined; other functions have no knowledge of them.

Using the program from the previous page:

```
1  #include <stdio.h>     // Include prototypes of printf and scanf
2
3  int square(int x);     // Prototype for the 'square' function
4
5  int main(void) {
6    int value = 2;
7    int valueSquared = square(value);
8
9    printf("%d\n", square(4));
10
11   return 0;
12 }
13
14 int square(int x) {
15    return x * x;
16 }
```

the compiler would give you an error if you did the following instead:

```
1  #include <stdio.h>     // Include prototypes of printf and scanf
2
3  int square(int x);     // Prototype for the 'square' function
4
5  int main(void) {
6    int value = 2;
7    int valueSquared = square(value);
8
9    square(4));
10   printf("%d\n", x);     // printing the value of x
11   return 0;
12 }
13
14 int square(int x) {
15    return x * x;
16 }
```

because `main()` knows nothing about a variable named x.

What would happen with the following?

```
1  #include <stdio.h>     // Include prototypes of printf and scanf
2
3  int square(int x);     // Prototype for the 'square' function
4
5  int main(void) {
6    int value = 2, x;    // also declaring x here
7    int valueSquared = square(value);
8
9    square(4));
10   printf("%d\n", x);
11   return 0;
12 }
13
14 int square(int x) {
15    return x * x;
16 }
```

7.6 Functions & Arrays

As with ordinary variables, it is also possible to pass the value of an array element, or an entire array, as an argument to a function.

The following is an example of passing an array element:

```
sq_root_result = squareRoot (averages[i]);
```

where the *i*th element of the array called `averages` is passed to the `squareRoot` function and the result is assigned to the variable called `sq_root_result`.

Passing an entire array as an argument is different – only the array name is required without any subscripts. For example, if you have an array called `gradeScores` that has been declared to have 100 elements:

```
int gradeScores [100];
```

it may be passed as an argument to a function:

```
minimumGrade = minimum (gradeScores);
```

and the function to which it is passed must be expecting an entire array to be passed as an argument. So the `minimum` function might look like the following:

```
int minimum ( int values[100] ) {
      …
      return minValue;
}
```

In the function header, the size of the formal parameter array is not needed; if it is specified, it will be ignored. So the above `minimum` function header would also be correct written as:

```
int minimum ( int values[] ) {
```

Unlike other arguments passed to functions, when an entire array is passed as an argument to a function, any changes made to the formal parameter array by the function are actually made to the original array passed to the function. Therefore, when the function returns, these changes still remain in effect. (When an array is passed as an argument, the function gets passed information describing *where* in the computer's memory the array is located.) This only applies to entire arrays, not individual elements of an array, whose values are copied into the formal parameters and therefore cannot be permanently changed by the function.

It is common to also send the size of the array as an argument as well:

```
int minimum (int values[], int numberOfElements ) {
```

This allows for the same function to be used for arrays of arbitrary length.

With multi-dimensional arrays, the same applies – you can pass the entire array by just using the name of the array as the argument, and any changes made to the formal parameter array inside the function makes permanent changes to the original array.

When declaring a 2-dimensional array, the number of rows in the array can be omitted, but the declaration *must* contain the number of columns in the array. So the following declarations are valid:

```
int array_values [100][50];
int array_values2 [ ][50];
```

The following 2 declarations are *not* valid:

```
int array_values [100][ ];
int array_values2 [ ][ ];
```

You can also have functions that accept variable-length multi-dimensional arrays as arguments. For example:

```
void displayMatrix ( int nRows, int nCols, int matrix[nRows][nCols] ) {
      …
}
```

7.7 Global Variables

In some cases, you do want a variable to be accessed by more than one function. This type of variable, a ***global variable***, would be declared outside of the functions, including `main` at the top of the file, right below your `#include` statements. These *global variables* then can be referenced directly by any function (i.e. they do not need to be passed to the function). This means that the value of a *global variable* exists for all the functions – there are not a bunch of different values for all the copies of the variable locally within each function. Global variables have a default initial value of zero (0), whereas, as we already know, local variables have no default initial value, so they must be explicitly initialized by the program.

It is more common to have global constants, rather than regular global variables. Since the value of global variables can be changed in any function, larger programs become harder to maintain. The use of global variables (that are not constants) is not allowed in this course.

7.8 Automatic & Static Variables

Recall that variables declared inside a function are *local variables* – more specifically, they are ***automatic local variables***. They are created each time the function is called, and as soon as the function is finished, they disappear. So, the values of those automatic local variables do not exist after the function completes execution.

However, if you place the word ***static*** in front of the variable declaration, then the value will remain even after the function completes execution. So, the next time the function is called, that ***static variable*** will still exist and have the value it had the last time that function executed. *Static variables* are initialized once, at the beginning of execution, and they have default initial values of zero (0).

The program on the next page demonstrates global variables as well as automatic and static variables.

```c
/*
 *  This program illustrates the use of static keyword,
 *  as well as automatic local variables, global variables,
 *  and variable shadowing
 */

#include <stdio.h>

int increment(int x);   // prototype
int number = 3;         // global variable


int main(void) {
    int i, x = 2;

    for (i = 0; i < 5; i++) {
        increment(x);
    }

    printf("number from main is %i\n", number);
    printf("x from main is %i\n", x);

    number = increment(x);

    printf("number after call to increment is %i\n", number);
    printf("x after call to increment is %i\n", x);

}



int increment(int x) {
    static int number;  // static variables have a default initial value of 0
                        // if not initialized to some other value

    number++;
    x += 5;

    printf("number in increment is %i\n", number);
    printf("x in increment is %i\n", x);
    return number;
}
```

7.8 Recursive Functions

Functions that call themselves are called ***recursive*** functions. They are commonly used in applications in which the solution to a problem can be expressed in terms of successively applying the same solution to subsets of the problem. The factorial example is the perfect example of this, where n! = n * (n – 1)!. This is recursive because the value of a factorial is based on the value of another factorial.

```
/* Program 7.17, page 158-159
     Calculating Factorials Recursively
*/

#include <stdio. h>

unsigned long int factorial (unsigned int n);   // prototype


int main ( void) {
     unsigned int j;

     for ( j = 0; j < 11; ++ j )
          printf ("% 2u! = % lu\ n", j, factorial ( j));

     return 0;
}


// Recursive function to calculate the factorial of a positive integer
unsigned long int factorial (unsigned int n) {
     unsigned long int result;

     if ( n == 0 )
          result = 1;
     else
          result = n * factorial ( n - 1);

     return result;
}
```