Chapter 5: Functions, Variables and Prototypes

OUR FIRST USER DEFINED FUNCTION

Load and examine the file sumsqres.c for an example of a C program with functions. Actually this is not the first function we have encountered because the main program we have been using all along is technically a function, as is the **printf()** function. The **printf()** function is a library function that was supplied with your compiler.

Notice the executable part of this program which begins in line 8 with a line that simply says "`header();`", which is the way to call any function. The parentheses are required because the C compiler uses them to determine that it is a function call and not simply a misplaced variable. When the program comes to this line of code, the function named **header()** is called, its statements are executed, and control returns to the statement following this call. Continuing on, we come to a **for** loop which will be executed 7 times and which calls another function named **square()** each time through the loop, and finally a function named **ending()** will be called and executed. For the moment ignore the variable name **index** in the parentheses of the call to **square()**. We have seen that this program calls a header, 7 square calls, and an ending. Now we need to define the functions.

DEFINING THE FUNCTIONS

Following the main program you will see another program beginning in line 14 that follows all of the rules set forth so far for a main program except that it is named **header()**. This is the function which is called from line 8 of the main program. Each of these statements are executed, and when they are all complete, control returns to the main program.

The first statement sets the variable named **sum** equal to zero because we plan to use it to accumulate a sum of squares. Since the variable named **sum** is defined prior to the main program, it is available for use in any of the functions which are defined after the variable is defined. It is called a global variable, and its scope is the entire program including all functions. It is also sometimes referred to as a file variable because it is available throughout the file. More will be said about the scope of variables near the end of this chapter. The statement in line 17 outputs a header message to the monitor. Program control then returns to the main program since there are no additional statements to execute in this function. Essentially, we drop out of the bottom of the function and return to the caller.

It should be clear to you that the two executable lines from this function could be moved to the main program, replacing the header call, and the program would do exactly the same thing that it does as it is now written. This does not minimize the value of functions, it merely illustrates the operation of this simple function in a simple way. You will find functions to be very valuable in C programming.

PASSING A VALUE TO A FUNCTION (CLASSIC METHOD)

Going back to the main program, and the **for** loop specifically, we find the new construct from the end of the last lesson used in the last part of the **for** loop, namely the `index++` used in line 9. You should get familiar with this construct, as you will see it in a lot of C programs.

In the call to the function named **square()**, we have an added feature, the variable name **index** within the parentheses. This is an indication to the compiler that when you jump to the function, you wish to take along the value of **index** to use in the execution of that function. Looking ahead at the function named **square()** in line 20, we find that another variable name is enclosed in its parentheses, the variable number. This is the name we prefer to call the variable passed to the function when we are in the function. We can call it anything we wish as long as it follows the rules of naming an identifier. Since the function must know what type the variable is, it is defined following the function name but before the opening brace of the function itself. Therefore, line 21 containing the expression "`int number;`" tells the function that the value passed to it will be an integer type variable. With all of that out of the way, we now have the value of **index** from the main program passed to the function **square()**, but renamed **number**, and available for use within the function. This is the classic style of defining function variables and has been in use since C was originally defined. A newer, and much better, method is gaining in popularity due to its many benefits and will be discussed later in this chapter.

Following the opening brace of the function, we define another variable named **numsq** for use only within the function itself, (more about that later) and proceed with the required calculations. We set the variable named **numsq** equal to the square of the value of number, then add **numsq** to the current total stored in the variable named **sum**. You should remember that the expression "`sum += numsq;`" has the same meaning as "`sum = sum + numsq;`" from the last lesson. We print the number and its square in line 27, and return to the main program.

MORE ABOUT PASSING A VALUE TO A FUNCTION

When we passed the value of the variable named **index** to the function, a little more happened than meets the eye. We didn't pass the variable named **index** to the function, we actually passed a copy of the value. In this way the original value is protected from accidental corruption by the called function. We could have modified the variable named **number** in any way we wished in the function named **square(),** and when we returned to the main program, the variable named **index** would not have been modified. We thus protect the value of a variable in the calling function from being accidentally corrupted, but we cannot return a value to the calling function from a called function using this technique. We will find a well defined method of returning values to the main program or to any calling function when we get to arrays and another method when we get to pointers. Until then, the only way you will be able to communicate back to the calling function will be with global variables. We have already hinted at global variables above, and will discuss them in detail later in this chapter.

Continuing in the main program, we come to the last function call, the call to the function named **ending()** in line 11. This line calls the last function which has no local variables defined. It prints out a message with the value of the variable **sum** contained in it to end the program. The program ends by returning to the main program and finding nothing else to do. Compile and run this program and observe the output.

NOW TO CONFESS A LITTLE LIE

I told you a short time ago that the only way to get a value back to the main program was through use of a global variable, but there is another way which we will discuss after you load and display the program named squares.c. In this example program we will see that it is simple to return a single value from a called function to the calling function. But once again, it is true that to return more than one value, we will need to study either arrays or pointers.

In the main program, we define two integers and begin a **for** loop in line 6 which will be executed 8 times. The first statement of the for loop is "`y = squ(x);`", which is a new and rather strange looking construct. From past experience, we should have no trouble understanding that the **squ(x)** portion of the statement is a call to the function named **squ()** taking along the value of **x** as a variable. Looking ahead to line 15 of the function itself, we find that the function prefers to call the input variable **input**, and it proceeds to square the value of **input** and call the result **square**. Finally, a new kind of a statement appears in line 21, the **return** statement. The value within the parentheses is assigned to the function itself and is returned as a usable value in the main program. Thus, the function call "`squ(x)`" is assigned the value of the square and returned to the main program such that the variable named **y** is then set equal to that value. If the variable named **x** were therefore assigned the value 4 prior to this call, **y** would then be set to 16 as a result of the code in line 7.

Another way to think of this is to consider the grouping of characters **squ(x)** as another variable with a value that is the square of **x**, and this new variable can be used any place it is legal to use a variable of its type. The values of the variables **x** and **y** are then printed out.

To illustrate that the grouping of **squ(x)** can be thought of as just another variable, another for loop is introduced in line 11 in which the function call is placed in the **printf()** statement rather than assigning it to a new variable.

One last point must be made, the type of variable returned must be defined in order to make sense of the data, but the compiler will default the type to **int** if none is specified. If any other type is desired, it must be explicitly defined. How to do this will be demonstrated in the next example program. We are simply using the default return value in this program.

Be sure to compile and run this program which also uses the classic method of defining function variables. Once again, any warnings can be ignored.

FLOATING POINT FUNCTIONS

Load the program floatsq.c for an example of a function with a floating point type of return. It begins by defining a global floating point variable named **z** which we will use later. Then in the main part of the program, an integer is defined, followed by two floating point variables, and then by two strange looking definitions. The expressions **sqr()** and **glsqr()** look like function calls and they are. This is the proper way in C to define that a function will return a value that is not of type **int**, but of some other type, in this case float. This tells the compiler that when a value is returned from either of these two functions, it will be of type **float**. This is, once again, the classic method of defining functions and is all but obsolete now.

Now refer to the function named **sqr()** starting in line 22 and you will see that the function name is preceded by the keyword **float**. This is an indication to the compiler that this function will return a value of type **float** to any program that calls it. The type of the function return is now compatible with the call to it. The line following the function name contains `float inval;`, which indicates to the compiler that the variable passed to this function from the calling program will be of type **float**.

The function named **glsqr()** beginning in line 31, will also return a **float** type variable, but it uses a global variable for input. It does the squaring right within the return statement and therefore has no need to define a separate variable to store the product.

The overall structure of this program should pose no problem and will not be discussed in any further detail. As is customary with all example programs, compile and run this program.

THE CLASSIC STYLE

The three programs we have studied in this chapter so far use the classic style of function definition. Although this was the first style defined for C, it is rapidly being replaced with a more modern method of function definition because the modern method does so much for you in detecting and flagging errors. As you read articles on C, you will see programs written in the classic style, so you need to be capable of reading them. This is the reason the classic style was included in this chapter. It would be highly recommended, however, that you learn and use the modern method which will be covered shortly in this tutorial. In fact, you are advised to never use the classic style for any of your programming efforts.

The remainder of this tutorial will use the modern method as recommended and defined by the ANSI standard. If you have an older compiler, it may not work on some of these files and it will be up to you to modify the programs as needed to conform to the classic style. Actually, the ANSI standard is used so universally, if you have a non-ANSI compiler you should use it only as a doorstop and purchase a good ANSI compatible compiler for the rest of your studies.

SCOPE OF VARIABLES

Load the next program, scope.c, and display it for a discussion of the scope of variables in a program. You can ignore the 4 statements in lines 2 through 5 of this program for a few moments. We will discuss them later.

The variable defined in line 7 is a global variable named **count** which is available to any function in the program since it is defined before any of the functions. It is always available because it exists during all the time that the program is being executed. (That will make sense shortly.) Farther down in the program, another global variable named **counter** is defined in line 25 which is also global but is not available to the main program since it is defined following the main program. A global variable is any variable that is defined outside of any function. Note that both of these variables are sometimes referred to as external variables because they are external to any functions, and they are sometimes also called file variables.

Return to the main program and you will see the variable named **index** defined as an integer in line 11. Ignore the word **register** for the moment. This variable is only available within the main program because that is where it is defined. In addition, it is an automatic variable, which means that it only comes into existence when the function in which it is contained is invoked, and ceases to exist when the function is finished. This really means nothing here because the main program is always in operation, even when it gives control to another function. Another integer is defined within the **for** loop braces named **stuff**. Any pairing of braces can contain variable definitions which will be valid and available only while the program is executing statements within those braces. The variables will be automatic variables and will cease to exist when execution leaves the braces. This is convenient to use for a loop counter or some other very localized variable.

MORE ON AUTOMATIC VARIABLES

Observe the function named **head1()** in line 26 which looks a little funny because of **void** being used twice. The purpose of the use of the word **void** will be explained shortly. The function contains a variable named **index**, which has nothing to do with the variable named **index** in line 11 of the main program, except that both are automatic variables. When the program is not actually executing statements in this function, this variable named **index** does not even exist. When **head1()** is called, the variable is generated, and when **head1()** completes its task, the variable in **head1()** named **index** is eliminated completely from existence. (The automatic variable is stored on the stack. This topic will be covered later.) Keep in mind however that this does not affect the variable of the same name in the main program, since it is a completely separate entity.

Automatic variables therefore, are automatically generated and disposed of when needed. The important thing to remember is that from one call of a function to the next call, the value of an automatic variable is not preserved and must therefore be reinitialized.

WHAT ARE STATIC VARIABLES ?

An additional variable type must be mentioned at this point, the static variable. By putting the keyword **static** in front of a variable definition within a function, the variable or variables in that definition are static variables and will stay in existence from call to call of the particular function. A static variable is initialized once, at load time, and is never reinitialized during execution of the program.

By putting the **static** keyword in front of an external variable, one outside of any function, it makes the variable private and not accessible to use in any other file. (This is a completely different use of the same keyword.) This implies that it is possible to refer to external variables in other separately compiled files, and that is true. Examples of this usage will be given in chapter 14 of this tutorial.

USING THE SAME NAME AGAIN

Refer to the function named **head2()**. It contains another definition of the variable named **count**. Even though **count** has already been defined as a global variable in line 7, it is perfectly all right to reuse the name in this function. It is a completely new variable that has nothing to do with the global variable of the same name, and causes the global variable to be unavailable within this function. This allows you to write programs using existing functions without worrying about what names were used for global variables or in other functions because there can be no conflict. You only need to worry about the variables that interface with the functions.

WHAT IS A REGISTER VARIABLE ?

Now to fulfill a promise made earlier about what a **register** variable is. A computer can keep data in a register or in memory. A register is much faster in operation than memory but there are very few registers available for the programmer to use. If there are certain variables that are used extensively in a program, you can designate that those variables are to be stored in a register in order to speed up the execution of the program. The method of doing this is illustrated in line 11. Your compiler probably allows you to use one or more register variables and will ignore additional requests if you request more than are available. The documentation for your compiler should list how many registers are available with your compiler. It will also inform you of what types of variables can be stored in a register. If your compiler does not allow the use of register variables, the register request will simply be ignored.

WHERE DO I DEFINE VARIABLES ?

Now for a refinement on a general rule stated earlier. When you have variables brought into a function as arguments to the function, and you are using the classic style of programming, they are defined immediately after the function name and prior to the opening brace for the executable statements. Local variables used in the function are defined at the beginning of the function, immediately following the opening brace of the function, and before any executable statements.

WHAT IS PROTOTYPING ?

A prototype is a model of a real thing and when programming with a good up-to-date C compiler, you have the ability to define a model of each function for the compiler. The compiler can then use the model to check each of your calls to the function and determine if you have used the correct number of arguments in the function call and if they are of the correct type. By using prototypes, you let the compiler do some additional error checking for you. The ANSI standard for C contains prototyping as part of its recommended standard. Every good C compiler will have prototyping available, so you should learn to use it. Much more will be said about prototyping throughout the remainder of this tutorial.

Returning to lines 3, 4, and 5 in scope.c, we have the prototypes for the three functions contained within the program. The first **void** in each line tells the compiler that these particular functions do not return a value, so that the compiler would flag the statement `index = head1();` as an error because nothing is returned to assign to the variable named **index**. The word **void** within the parentheses tells the compiler that this function requires no parameters and if a variable were included, it would be an error and the compiler would issue a warning message. If you wrote the statement `head1(index);`, it would be a error. This allows you to use type checking when programming in C in much the same manner that it is used in Pascal, Modula-2, or Ada, although the type checking in C is relatively weak.

Note the addition of the word **void** in line 9. This is an indication to the system that we do not plan to return a value to the operating system when we terminate operation of this program. The main program also can return an **int** value in the same manner as any other function. This value is returned to the operating system where it can be ignored or it's value used as an indication of what the program did.

You may have been getting warning messages when you compiled many of the example programs in this tutorial, and you now see the reasons for these warnings. The compiler was telling you that it did not know how to handle the lack of a return definition. This will be fixed in all of the remaining example programs in this tutorial.

You should begin using prototype checking at this time, if it is available with your compiler. Your compiler may have an option that will require a prototype for every function. This should be enabled and left enabled. Check your documentation for the details of how to do it. Prototyping will be used throughout the remainder of this tutorial. If your compiler does not support prototyping and the modern method of function definition, you will have to modify the remaining example programs. A much better solution would be to purchase a better compiler.

Line 2 of scope.c tells the system to go to the standard directory where include files are stored and get the file named **stdio.h** which contains the prototypes for the standard input and output functions so they can be checked for proper variable types. Don't worry about the include yet, it will be covered in detail later in this tutorial. Be sure to compile and execute this program. If you have been getting warnings about the function **printf()** not being defined, it is fixed by including the stdio.h file as we have done here. We will explain why this fixes the warnings shortly.

STANDARD FUNCTION LIBRARIES

Every compiler comes with some standard predefined functions which are available for your use. These are mostly input/output functions, character and string manipulation functions, and math functions. We will cover most of these in subsequent chapters. Prototypes are defined for you by the writer of your compiler for all of the functions that are included with your compiler. A few minutes spent studying your reference guide will give you an insight in where the prototypes are defined for each of the functions. Most compilers have additional functions predefined that are not standard but allow the programmer to get the most out of his particular computer. In the case of the IBM-PC and compatibles, most of these functions allow the programmer to use the BIOS services available in the operating system, or to write directly to the video monitor or to any place in memory. These will not be covered in any detail as you will be able to study these unique aspects of your compiler on your own. Several of these kinds of functions are used in the example programs in chapter 14.

WHAT IS RECURSION ?

Recursion is another of those programming techniques that seem very intimidating the first time you come across it, but if you will load and display the example program named recurson.c, we will take all of the mystery out of it. This is probably the simplest recursive program that it is possible to write and it is therefore a stupid program in actual practice, but for purposes of illustration, it is excellent.

Recursion is nothing more than a function that calls itself. It is therefore in a loop which must have a way of terminating. In the program on your monitor, the variable named **index** is set to 8 in line 9, and is used as the argument to the function named **count_dn()**. The function simply decrements the variable, prints it out in a message, and if the variable is not zero, it calls itself, where it decrements the variable again, prints it, etc. etc. etc. Finally, the variable will reach zero, and the function will not call itself again. Instead, it will return to the prior time it called itself, and return again, and again, until finally it will return to the main program and from there return to DOS.

For purposes of understanding you can think of it as having 8 copies of the function named **count_dn ()** available and it simply called all of them one at a time, keeping track of which copy it was in at any given time. That is not what actually happened, but it is a reasonable illustration for you to begin understanding what it was really doing.

WHAT DID IT DO ?

A better explanation of what actually happened is in order. When you called the function from itself, it stored all of the variables and all of the internal flags it needs to complete the function in a block somewhere. The next time it called itself, it did the same thing, creating and storing another block of everything it needed to complete that function call. It continued making these blocks and storing them away until it reached the last function when it started retrieving the blocks of data, and using them to complete each function call. The blocks were stored on an internal part of the computer called the stack. This is a part of memory carefully organized to store data just as described above. It is beyond the scope of this tutorial to describe the stack in detail, but it would be good for your programming experience to read some material describing the stack. A stack is used in nearly all modern computers for internal housekeeping chores.

In using recursion, you may desire to write a program with indirect recursion as opposed to the direct recursion described above. Indirect recursion would be when a function A calls the function B, which in turn calls A, etc. This is entirely permissible, the system will take care of putting the necessary things on the stack and retrieving them when needed again. There is no reason why you could not have three functions calling each other in a circle, or four, or five, etc. The C compiler will take care of all of the details for you.

The thing you must remember about recursion is that at some point, something must go to zero, or reach some predefined point to terminate the loop. If not, you will have an infinite loop, and the stack will fill up and overflow, giving you an error and stopping the program rather abruptly.

ANOTHER EXAMPLE OF RECURSION

The program named backward.c is another example of recursion, so load it and display it on your screen at this time. This program is similar to the last one except that it uses a character array. Each successive call to the function named **forward_and_backwards()** causes one character of the message to be printed. Additionally, each time the function ends, one of the characters is printed again, this time backwards as the string of recursive function calls is retraced.

This program uses the modern method of function definition and includes full prototype definitions. The modern method of function definition moves the types of the variables into the parentheses along with the variable names themselves. The final result is that the line containing the function definition looks more like the corresponding line in a language with relatively strong type checking such as

Pascal, Modula-2, or Ada. The prototype in line 5 is simply a copy of the function header in line 18 followed by a semicolon. The designers of C even allow you to include a variable name along with each type. The name is ignored by the compiler but including the name in the prototype could give you a good idea of how the variable is used, acting like a comment.

Don't worry about the character array defined in line 9 or the other new material presented here. After you complete chapter 7 of this tutorial, this program will make sense. It was felt that introducing a second example of recursion was important so this file is included here.

Compile and run this program with prototype checking enabled and observe the results.

THE FLOAT SQUARE PROGRAM WITH PROTOTYPES

Load and display the program named floatsq2.c which is an exact copy of the program floatsq.c which we considered earlier with prototyping added. The use of prototyping is a good practice for all C programmers to get into.

Several things should be mentioned about this program. First, the word **float** at the beginning of lines 27 and 35 indicate to the compiler that these functions are functions that return **float** type values. Also, since the prototypes are given before main, the functions are not required to be identified in line 12 as they were in line 7 of floatsq.c earlier in this chapter. Notice also that the type of the variable named **inval** is included within the parentheses in line 27.

Now that we understand prototypes, we can eliminate most of the warnings we have been getting during our study of this tutorial. It would be good practice for you to eliminate all warnings in all of your programs, since the compiler is trying to warn you of some potentially bad programming constructs.

MORE STYLE ISSUES

The example named style2.c is given as an illustration of various ways to format a function. You will note different ways to define the input parameters. Examples three and four are both the same style, but example four illustrates the style when nothing is passed in or returned. This style states very clearly that nothing is needed or returned and it cannot be construed as an oversight. Spend some time studying these function examples, then begin developing the style you will use. If you are like most programmers, you will develop a style that you plan to use forever, then change it every few months or on every new project.

Programming Exercise:

---

1. Rewrite tempconv.c, from an earlier chapter, and move the temperature calculation to a function.
2. Write a program that writes your name on the monitor 10 times by calling a function to do the writing. Move the called function ahead of the main function to see if your C compiler will allow it.
3. Add prototyping to the programs named sumsqres.c and SQUARES.C, and change the function definitions to the modern method.

The Webwizard