Lesson 5 Using Functions in C

Introduction

The use of functions in programming derives from the idea of breaking a program down into sub programs. By using functions or sub-routines, we can avail of the following positives.

Firstly, as the code below should demonstrate, we can avoid unnecessary repetition of code. The function drawbox() is only written once but is used twice.

Secondly, by using functions to carry out specific tasks, we can organise our program in a way that makes it easier to read and comprehend.

Thirdly, independence. Functions can have there own private variables that cannot be accessed from outside of the function. This means that a programmer need not worry about using the same variable name in different functions and also they can debug smaller units of code.

In C, functions can be made return a value by a return statement. Functions can accept a value via arguments. Compile and run the following code.

```
#include<stdio.h>

void drawbox(void); /* prototype */

 int main(void)
 {
  drawbox();
  printf(" Welcome to C ");
  drawbox();
  return(0);
 }

 void drawbox(void) /* declaration */
  {
   int k;
   printf("\n####################");
   for(k=1;k<=5;k++)
   printf("\n#\t\t #");
   printf("\n####################\n");
  }
```

This program draws a box around the words “Welcome to C”. The function drawbox() is used to draw the top and the bottom of the box. We can see here that the code was written once but implemented twice. To change the height and width of the box we only need to modify the code inside the function drawbox().

Structure of a function

The order of where the function code appears in a file is not important. It may even be called from another file.

Note that main() is also a function. The function main() will always be executed first and is considered the entry point to a program.

When writing a function, there are three program elements to consider. Firstly, the function **definition**. This is the actual function code itself.

The line void *drawbox(void)* is the **declarator**. In this example, the function drawbox() has no arguments and is set to return nothing, of type void. Following this line the body of the function is defined. The body is enclosed in braces.

The function **prototype** is written before main(). The function prototype tells the compiler the name of the function, the data type it returns (if any) and the arguments which are passed to it (if any). Finally a function **call** is needed to execute the function.

How does this differ to a bash function ?

Local variables

The variable k used in the function drawbox() is called a local variable. We could define another variable k in function main() (recall main() is also a function) which will be different from the k in drawbox().

A local variable used in this way is often called an automatic variable. Its created when a function is called and destroyed when a function returns. Its scope or visibility or lifetime is confined to the function. When the function is finished the variable dies. When the function gets called again, it recreated.

Function returning values

In the following code below, the function calculates the square root of a number using C`s own pow() function from its maths library. Note 10 to the power of 0.5 is the same as square root 10. The maths library needs to get a mention as part of the compiling process ie gcc mycode.c -lm

```
#include<stdio.h>
#include<math.h>
double square_root(void); /* prototype */
int main()
{
   printf("square root of 10 = %e \n",square_root());
   return(0);
}
double square_root(void)
{
 double num=10.0;
 return(pow(num,0.5));
 }
```

Thee function has no arguments but returns a double. The return statement has two purposes. When executed, it returns something back to the calling program. Whatever it contains inside the return parentheses will be returned as a value to the calling program. The return statement suffers from one serious drawback, you can only use it to return one value from a function.

You may notice in the above code the use of the type double.

Passing data as an argument to a function

Suppose that we wish to write a function that returns the square root of any number we pass to it. This can be done as follows.

Note in this example the user enters a number that is stored as a float, but the function returns a double. Once again you will need to link in the maths library -lm.

```
#include<stdio.h>
#include<math.h>
double square_root(float);

int main()
   {
    float num;
    printf("Enter he number you wish to find the square root of");
    scanf("%f",&num);
     printf("square root of 10 = %e \n",square_root(num));
     return(0);
   }

double square_root(float hold)
   {
    return(pow(hold,0.5));
   }
```

The above program is passed one argument down from the calling program. We can however pass any number of arguments to a function. Consider the following program which draws a rectangle which represents the ground plan of a room. It receives two arguments, the length and the width.

```
#include<stdio.h>
void draw_rect(int,int)
int main(void)
 {
   printf("\nSITTINGROOM\n");
   draw_rect(22,14);
   printf("\nBATHROOM\n");
   draw_rect(6,6);
   printf("\nBEDROOM \n");
   draw_rect(10,8);
   return(0);
  }

void draw_rect(int x,int y)
    {
     int i,j;
     x/=2;y/=2;            /*scaling factor*/
    for(i=1;i<=x;i++)
        {
         printf("\t\t");
          for(j=1;j<=y;j++)
            printf("\|");
         printf("\n");
        }
}
```

The structure of this program is very similar to the previous one except that now two arguments are passed to the function. This can be extended to any number of arguments.

Communications between functions

In C, we can use as many functions as we like in our code. A function can call another function. All functions, including main are visible to all other functions, they all have equal status. As an exercise write a program with three functions which finds the difference of two squares. One function should square a number, another function should find the difference of two numbers and the third should return the difference of squares of the two arguments.

K and R style and ANSI C

The original C language by Kernigan and Ritchie - "The C Programming Language - Second Edition" is the definitive text of the classic style of C programming. However they did not include function prototypes. Prototypes are a language refinement that was introduced with the ANSI standard. They should used when programming.

When prototyping was added to the language, many programmers apparently thought that the **main()** function didn't return anything, so used the **void** type as a return. Thus it became a common practice to write the **main()** function as follows;

```
void main()
{
   ...
}
```

When the ANSI C standard was finalized the only return type approved by the standard was an **int** type variable. The compiler should check that the program actually returns a value by requiring that an integer is returned from each exit point. The correct form of the above becomes

```
int main()
{       ...
      return 0;
}
```

Apparently because of the inertia behind the use of a **void** return, many compiler writers added the **void** return as an extension to permit the use of legacy code without modifications. Some compilers therefore support the **void** return however the **int** return is the only method approved by the ANSI C standard.

In order to make your code as portable as possible, you should always use the ANSI version above.

The value of zero that is returned to the operating system indicates that the program executed normally.

External variables

So far, all the variables used have been visible only to the functions where they have been defined. These variables are called local or automatic variables. When the function is called, they are created and when the function is completed, they are deleted. Often we may wish to define variables that have greater scope so that all the functions can use them. In this case we can use external or global variables. Consider the program below.

```
#include<stdio.h>
int radius; /*external or global variable*/
float circle();       float sphere();
```

```
void main(void)
	{
	puts("ENTER THE RADIUS YOU WISH TO USE");
	scanf("%d",&radius);
	printf("\n THE CIRCLE OF RADIUS %d HAS AREA %f\n",radius,circle());
	printf("\n THE SPHERE OF RADIUS %d HAS VOL  %f\n",radius,sphere());
	return(0);
	}

float circle(void)
	{
	return(3.14*radius*radius);
	}
float sphere(void)
	{
	 return( (4.0/3.0)*(3.14)*(radius*radius*radius));
	}
```

In this program, the variable radius is visible and accessible from the functions main(), circle() and sphere(). Although you may be tempted to make all variables external, you should not do so. External variables are not protected from accidental alteration by functions. This makes some bugs harder to find. Also external variables use memory less efficiently than local variables. Thus unless necessary, you should always use local variables.

Static variables

By putting the keyword **static** in front of a variable definition within a function, the variable is said to be a static variable and will stay in existence from call to call of that particular function. A static variable is initialized once, at load time, and is never reinitialized during execution of the program. Thus the variable will not be recreated for each function call.

By putting the **static** keyword in front of an external variable, or outside of any function, it makes the variable private and not accessible for use in any other file. Note that this is a completely different use of the same keyword.

In all our examples so far, all the code has been in a single file. For larger applications it makes sense to split the code up into several files. For example, suppose you have a program that calculates the surface area and volume of different objects, one file may contain all the function definitions for volume calculations, a second file might contain the function definitions for area calculations and a third file may contain main(). Now the question of variable visibility across multiple files is valid.

The use of a static variable outside of main() in the third file means that the variable is not accessible from either file one or two.

Standard Function Libraries

Every C compiler comes with some standard predefined functions which are available for your use.

These maybe input/output functions, character and string manipulation functions and mathematical functions. Prototypes are defined for you for the different function. A few minutes spent studying your reference guide will give you an insight to where the prototypes are defined for each of the functions.

Try to use standard ANSI C functions to ensure that your code is as portable as possible.

Programming Exercises

1. Write a short program that uses a function to print out your name in a box when called.

2. Write a program that prints out the smaller of two numbers entered form the keyboard. Use a function to do the actual comparison of the two numbers.

3. Write a program that contains two functions which return the volume of a cone and the volume of a hemisphere.
4. Write a program that reads in a persons exam score and produces a horizontal bar chart of the results (6 students). Use a function to draw the graphics on the screen. This function should be of type void and have the students mark as its argument. *(HINT for(i=1;i<=score;i++) printf("#");)*

5. Write a mathematical function which determines the exponential of a number using exp(x) = 1 +

   x / 1! + x^2 / 2! + x^3 / 3! + x^4 / 4! + …

   Compare your result for different x with the standard exp() function.

6. Write a program with separate functions which determines the slope of a straight line and the point at which the line crosses the Y-axis. Complete the following table

x1	y1	x2	y2	slope	intercept
3	3	4	5	2	-3
-1	5	0	-1		
100	50	-10	-10		
1	-1	1	3		