Chapter 5 – Functions

By the end of this chapter you should:

- Understand how to declare & use subroutines in C
- Know the difference between a function & a procedure

Key definitions to ensure you know:

- Function
- Procedure
- Sub-routine
- Global variable
- Local variable
- Argument
- Parameter
- Function prototype
- Pointer
- **Passing by reference**
- **Passing by value**

Why use subroutines?

- Long blocks of code get hard to read, debug and fix
- Often subroutines can be re-used across lots of bits of code, which saves time
- Breaking code up into subroutines also makes it easier for teams of programmers to work together, sharing the work out, customising it based on expertise
- Most programming languages allow you to break up your code into subroutines – which can be defined as procedures and functions
- These subroutines are smaller programs that can be 'called' (jumped to) by the main program to do a specific task
- A subroutine is usually used to do ONE specific task and should be called by a name that makes it easy to remember what it does i.e. a function to read in ages should be called something like **read_age()**

Key points

- A function MUST have a unique name – just like a variable. By using this name in the main program or within other functions you can call the function and execute the code contained within it.
- A function is independent - it can perform its task without interference from other parts of the program.
- A function can accept and return a value from the program that calls it e.g.
  - you can pass one or more variables into a function, and it can return ONE value to the program that called it
- **`printf()`** and **`scanf()`** are functions we already know, they're built into C and so we don't ever see the code that runs when we use them

This chapter….

- Needs to introduce you to how to use subroutines properly and shows you a range of examples of the below variations of how subroutines can be used.
- In C, all subroutines are called functions BUT:
  - A C function can be declared as **void** – this means it will not return a value. In other coding languages this type of subroutine is called a **procedure….**
  - **….**OR it is declared with a data type e.g. **int** or **float** – this means that it *has to* return ONE value of that type to the main program. This is a **function.**
- All subroutines can have none, one or many **parameters** (items of data passed into them)

An **integer function**

```
int squareNum (int x);
```

```
main()
{
   int input=0, answer=0;
   printf("Enter  a number:   ");
   scanf("%d", &input);
   answer = squareNum(input);
   printf("\n %d squared = %d. ", input, answer);
   printf("\n The value of x is: %d ", x)
}
```

```
int squareNum (int x)
{
          int x_squared;
          x_squared = x * x;
          return x_squared;
}
```

This is the function – more on the next page…

- This is the function prototype – it tells the compiler that there will be a subroutine, and defines **what data type it will return,** its **name** and any **parameters** it will take
- All your function prototypes need to be declared at the top of your code outside of the **main()** function. They DO have a semi-colon at the end
- This prototype declares a parameter **x**– this is an integer variable that will be passed to a function as an **argument** from the main program
- This is where the function is called (jumped to)– the variable **input** is being used as its argument which will be passed into the function itself and will be copied and stored in the **x** parameter
- **answer** is the variable that will hold the integer that will be returned from the function
- A function **ALWAYS** needs to be part of a variable assignment expression – this is because something has to hold the return value passed back from the function

Integer function continued….

```
int squareNum (int x);


main()
{
   int input=0, answer=0;
   printf("Enter  a number: ");
   scanf("%d", &input);
   answer = squareNum(input);
   printf("\n %d squared =%d.", input, answer);
   printf("\n The value of x is: %d ", x)
}

int squareNum (int x)
{
         int x_squared;
         x_squared = x * x;
         return x_squared;
}
```

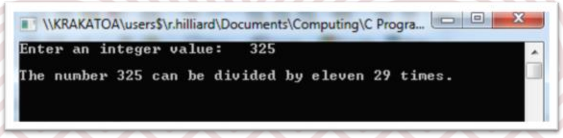
- The argument passed into the function is the variable **input**, which was typed in on the line before
- This is the function header: it MUST match the prototype (the bit declared at the top of the code) except WITHOUT the semi-colon at the end of the statement
- **x** is the **parameter**– this is where the argument **input** is sent when it is passed from the main program – it is **good practice** to make sure your arguments and parameters don't have the exact same names – it can get confusing during debugging otherwise!
- **x** and **x_squared** are both **local variables** – they only exist in the subroutine
- E.g. this line would throw an error as the main() subroutine doesn't know what x is!
- When the code in the subroutine has been carried out, the local variable **x_squared** is **returned** to the main program, where the function jumped from, and is therefore assigned to and stored in the variable **answer**

Exercise 1

Write a program that uses shows you can write a function, pass a parameter and return a value.

Requirements:

- The program should allow a user to enter a number
- It should then pass the number as a parameter to an int function called **divEleven()**
- The function should then calculate how many times the input number can be divided by 11
- The result of the sum should be returned to the main program and output to screen e.g.


\\KRAKATOA\users$\r.hilliard\Documents\Computing\C Progra...

```
Enter an integer value:   325
The number 325 can be divided by eleven 29 times.
```

void functions (procedures)

```
void enter_Array();
void get_Average();

int Scores[5];

main(){
   printf("Enter scores for 5 students:");
   enterArray();
   getAverage();
}

void enter_Array(){
   for (int x=0; x<5; x++){
          printf("Score %d:", x+1);
          scanf("%d", &Scores[x]);
   }

void get_Average(){
   float average=0;
   for (int y=0; y<5; y++){
          average= average + Scores[y];
   }
   average = average/5;
   printf("\n The average of the scores is: %f", average);
}
```

- Two function prototypes are declared at the top of the code
- These are the calls to the procedures (functions) – in this case neither function has a parameter and because they are void – **don't have a return value** – they can be called on a line on their own and don't need to be part of a variable expression
- **Both subroutines can use the Scores[] array data because it's been declared globally – arrays can be a pain to pass into a function and are often declared this way instead**

Exercise 2

Write a program that uses a sub-routine called `testCount()` which will use a loop to count up to a set value and show a message after each loop.

Requirements:

– The program should ask the user what number to count up to
– Call the `testCount()` function, passing the input as a parameter
– The function should use a suitable loop to count up to that value
– It should output a message to screen during each loop showing each increment of the number
– When the subroutine is complete, the main program should output a message stating "Void function complete".

Functions with >1 parameter

- So far we have only used functions that had none or one parameter, but you can pass two or more arguments into a function by adding them to the prototype's parameter list like this:

  int findVol **(float length, int width, int depth)**

- We've declared 3 parameters in the brackets, each with their own data type declaration
- Then, when we want to use call the function in the code, we would pass in the 3 arguments into it e.g.:

  volume = findVol(h, w, d);

Exercise 3

Edit the calculator program that you made for Exercise 3 back in Chapter 2 so that each case statement calls a separate function to carry out the chosen operation.

Requirements:

- The program should ask the user to enter two integers
- Each Case statement for +, -, * and / should then call a function `add()`, `sub()`, `mult()` or `div()`
- The two input numbers should be passed into the functions as parameters
- Each function should return its calculated value back to the `main()` function, where the result of the sum should be output

Passing arrays/ strings to a function

- This has to be done a little differently because an array isn't one piece of data and it isn't stored in one memory location
- Instead of actually **passing the value** into the function, what we do is **pass a reference** to where the array can be found in memory instead (which is basically like making the array a secure global variable – only a function that is passed the pointer can access it rather than any piece of code)
- When we pass by value the code makes a local copy of the variable in the new subroutine – so both still exist…..
- …..when we pass by reference, all pieces of code edit the same variable so all changes happen to the same piece of data
- To pass by reference, we add a **pointer** to the memory location of the data we are passing as a parameter. We do this by putting an asterisk before the variable/array name e.g.

```
int checkScores(int *Scores)
```

Passing a pointer - example

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

void revWord(char *revWord);

main () {
    char word[5];
    printf("Enter a 4 letter word:");
    scanf("%s", word);
    printf("Before reverse, word is : %s\n", word );
    revWord(word);
    printf("After reverse, word is : %s\n", word );
}

void revWord(char *revWord){
    char tempWord[5];
    int j=0;
    for(int i=3;i>=0;i--){
        tempWord[j]= revWord[i];
        j++;
    }
    tempWord[4]=revWord[4];
    strcpy(revWord,tempWord);
}
```

- **I have used a string function in my code to copy one string into another, so I had to include the string library** (more in the next chapter about those!)
- I call my revWord procedure with its argument, **`word`**, just like I did in the other examples in the chapter
- **The only difference is in the parameter declaration, where I have added a pointer - the * before the string name…..**notice that I haven't included the string bounds (the [5] part) in my parameter declaration
- **Copy this code and try it out** – make sure that you understand how passing by reference works – it's a key exam theory question as well as being really useful when coding!

Exercise 4

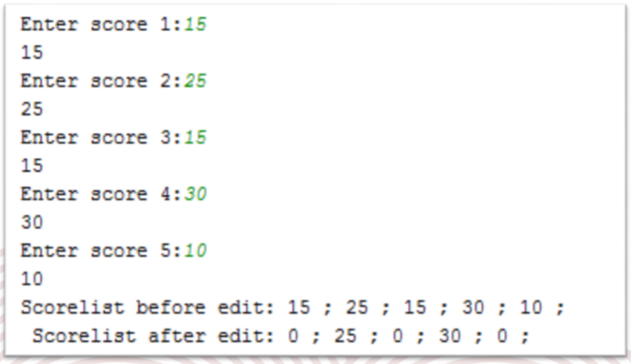
Write a program that will store 5 test scores, then edit the score list to blank out any scores that are 20 or less.

Requirements:

- In the **`main()`** function:
  - Ask the user to input 5 scores and store these into an array
  - The score should be between 0 and 75
  - Pass the scores as a parameter to a function named **`editScores()`**
  - Create a **`printScores()`** function to loop through and print the scores array – this will need to be passed in as a parameter
  - This function should be called BEFORE the scores are edited, and then again afterwards
- In **`editScores()`**:
  - Edit the data in the scores array so that any score of 20 or lower is set to 0

Example output on next page

Example output:



```
Enter score 1:15
15
Enter score 2:25
25
Enter score 3:15
15
Enter score 4:30
30
Enter score 5:10
10
Scorelist before edit: 15 ; 25 ; 15 ; 30 ; 10 ;
 Scorelist after edit: 0 ; 25 ; 0 ; 30 ; 0 ;
```