C-05 Functions in C

5.1 General

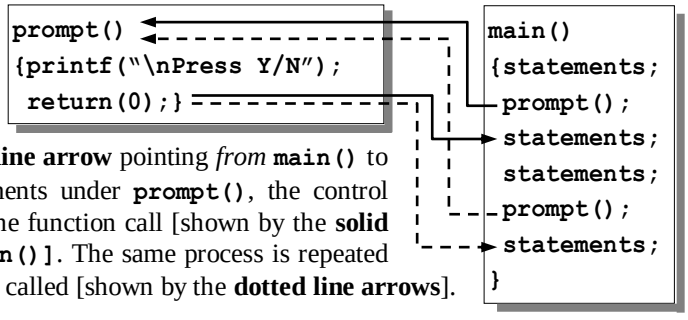
When a program becomes large, it may be difficult to organise everything within a single main function. Suppose the main function consists of several thousand lines of code. Within such a program it will be very difficult to locate any portion of the code that may need debugging or change and will be similar to finding a needle in a haystack. Moreover some portion of the code may have to be repeated at different points of the program leading to rewriting of the same code at the different points and hence unnecessarily increasing the size of the program. Under such circumstances, if the program could have been broken down into several separate compartments based on the type of job carried out by each section of the code, then it would have been easier to handle the program. The feature that C provides to do this type of coding is called a function.

A function is a block of instructions that together can perform a certain task. As had been mentioned at the beginning of the course, a C program in general consists of a number of such functions or modules of code, `main()` being the most important one. You had been already using several library functions to carry out such jobs as taking inputs from the keyboard [`getchar()`], printing output onto the screen [`printf()`], finding the square-root of a number [`sqrt()`] etc. These functions were available in the standard library provided by C and one had to just **call** these functions to execute them. Each of these functions had a unique identifiable purpose and executed a specific type of job as had been defined by the scope of that particular function. Thus to print something onto the screen at several points in a program, instead of writing the whole code at all those points, one just calls the `printf()` function to get the desired output. But it is highly unlikely that functions to do all types of jobs are available in the standard libraries. Every program has its own set of special or unique necessities leading to the use of some special functions, which may not be available in the standard libraries. We will now discuss the methods to define and use a user-defined function. Before doing that, let us write down the usefulness of using functions in a program.

- ***Repetitive jobs can be assigned to a function***: Any repetitive job that needs to be executed at several points of code can be assigned to a particular function. For example consider the sine series given by the relation $\sin(x) = x/(1!) - x^3/(3!) + x^5/(5!) - \ldots$ Here the factorial of a number is to be calculated at several points of the program and necessitates the use of a function called factorial which can be called each time a factorial needs to be calculated.
- ***Increases readability of a program***: A function consists of self-contained components i.e. it is self sufficient in nature. Thus if a program is compartmentalised using functions to carry out different jobs, it becomes easier to follow the program logic as one has to concentrate on a certain portion of the code only. This increases readability of a program.
- ***Reduces chances of error and helps in debugging***: Increase in readability also helps in pin pointing possible errors within a section of the code and accordingly helps in debugging.
- ***Helps in shared coding***: A large program consisting of several functions can be developed by several persons. Each person can code one or more functions independently and then all the modules can be later on put together to build the whole program. This makes writing of large programs faster as the whole job can be broken down into several logical parts, which can be simultaneously coded.
- ***Reuse of code***: A function carrying out a certain particular work can be reused in more than one programs by making libraries of such user-defined functions and calling them in the programs as and when required. This saves a lot of time unnecessarily rewriting the same code in separate programs.

5.2 The working of a Function:

Let us now see how a function functions. The following schematic diagram shows how a function called **`prompt()`** gets executed when being called from the **`main()`** function.



The function **`prompt()`** is been called two times from **`main()`**. When called for the first time, the program control *temporarily* goes from **`main()`** function to the called function i.e. **`prompt()`** [shown by the **solid line arrow** pointing *from* **`main()`** to **`prompt()`**]. After execution of the statements under **`prompt()`**, the control again comes back to the statement next to the function call [shown by the **solid line arrow** pointing *from* **`prompt()`** to **`main()`**]. The same process is repeated the next time the function **`prompt()`** is been called [shown by the **dotted line arrows**].

Though the above example is of no important use but it illustrates the way a function is used in a program. Instead of writing the code for the common task of printing on the screen “Press Y/N” at several points in the **main()** function, we have used a function called **prompt()** to do the same job for us. Whenever the comment is to be printed we just call the function **prompt()** and it executes the job for us. Later on if you want to do some extra thing using **prompt()**, you just add the extra lines inside the body of the function **prompt()** and it will work as per your requirement. On the other hand if you had not used a function to do the job and you needed a change at a later stage, then you would have to rewrite the changes at all the points where you had used that piece of code. Surely using a function is a much better option!

A function is similar in structure to the **main()** function. Just like the **main()** function it processes some information when requested for. The request to a function is referred to as **calling** the function. But before being called, the function needs to be *defined* first. A function can be defined in two ways. In the first method a **function prototype** is declared and then the function defined at a later stage. Defining a

```
include header files

prototype for function_1() declared;
prototype for function_2() declared;
… … …
main()
{  statements;
   function_1();
   statements;
   function_2();
   function_1();
   statements;
}

function_1() defined
{ statements for function_1; }

function_2() defined
{ statements for function_2; }
```

Fig. 1

```
include header files

function_1() defined
{ statements for function_1; }

function_2() defined
{ statements for function_2; }

main()
{  statements;
   function_1();
   statements;
   function_2();
   function_1();
   statements;
}
```

Fig. 2

prototype results in informing the compiler in advance during compilation that these are the function names that will be used by the **main()** function or other functions and are defined later on, so that when the compiler encounters the name of the functions, it knows in advance that these refer to functions that are defined later on. If the compiler encounters a function call before it has been declared or defined, then it causes a program error and displays the message that function prototype not found. **Fig. 1** shows the general structure of a program using functions whose prototypes are declared before the **main()** function and later the functions are defined after the **main()**. **Fig.2** shows the alternative method of defining the functions first and then using them in **main()**. In this case the compiler already has the definitions of the functions before they are being called by **main()** and hence does not cause any error during compilation.

Of the above two methods, the *first method is the preferred one* because in that case a function once declared as a prototype can be defined anywhere i.e. either *before* or *after* the **main()** function (but never inside the **main()** function, as a function definition *within* another function is forbidden). Whereas in the second method, the functions need to be defined before **main()**, keeping in mind the fact that the functions should be so arranged that a function is not been called before it has been defined. This is to be particularly noted for the case when a user-defined function calls another user-defined function. Thus in the above example if **function_1()** calls **function_2()**, then instead of defining **function_1()** first and then **function_2()**, we have to define **function_2()** first and then **function_1()**. But this situation can be completely avoided if function prototypes are used, as then being declared first it does not matter the order in which they are defined.

5.3 Defining a Function:

Irrespective of whether a function is declared using a prototype or not, let us see how a general function is defined. It consists of the data type of the return value of the function followed by the function name and then a list of variables with respective data types which are the function arguments placed inside brackets.

```
function_return_data_type  function_name(function argument/parameter list)
```

Ex.1: Let us take an example to see how a function is defined and used. In the following program an user-defined function called **power()** is used to find the value of x^y where x is a float, and y is an integer and both are input by the user. The values of x and y are entered by the user in the **main()** function and then they are utilised by the user-defined function to calculate the desired output.

```
#include<stdio.h>
float power(float base, int index)
{ float result=1;
  int i=1;
  for(;i<=index;i++)
  result*=base;
  return result;
}
main()
{ float x, output;
  int y;
  printf("\nProgram to Calculate x^y");
  printf("\nEnter base x (decimal value)");
  scanf("%f", &x);
  printf("\nEnter index y (integer value)");
  scanf("%d", &y);
  output=power(x,y);
  printf("\nValue of x^y is %f", output);
  return(0);
}
```

Statements within the function placed inside curly { } brackets

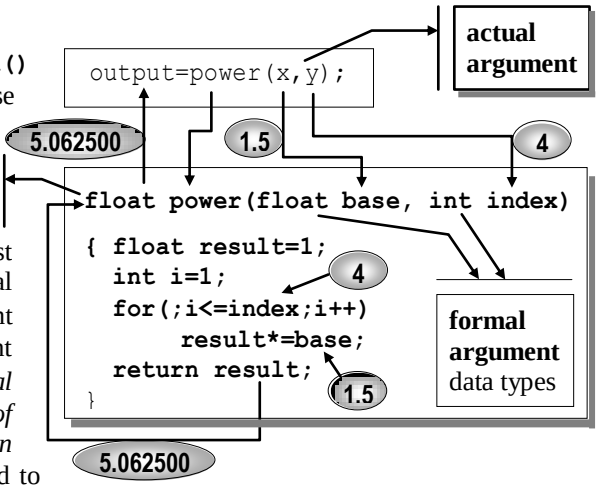
Definition of the Function called **power()**

The function `power()` called from here.

Let us now analyse how the function is working in the above example. First let us consider the **main()** function from where the program starts. The user inputs the two values: a **float** number **x** and an **integer** number **y**. After this the user-defined function **power()** is assigned to the float variable output. The assignment results in **calling** the user-defined function **power()**. But for the function **power()** to work, it should have some input from the calling function, i.e. it should know what to raise to the power of what. This information is provided by placing the variables **x** and **y** inside brackets after the name of the function **power()** in the calling function. These values that are passed to the called function from the calling function, are called the **actual arguments** of the function.

Once the variables to work with i.e. the actual arguments are passed to the called function, it should be able to receive them, otherwise these would be of no use. In the above example, the called function **power()** receives the actual arguments from the calling function **main()** in the two variables viz. a **float** variable **base** and an **int** variable **index**. These variables in the called function (i.e. base and index in this case) are called the **formal arguments** and are declared *inside the brackets* following the name of the function.

Thus the formal arguments of a called function are the variables, which receive the actual arguments from the calling function.



Let us input the values **x=1.5** and **y=4** in the **main()** function. The figure to the right shows how these values are transferred to the function **power()**. Note that the formal arguments are placed in the **same order** inside the function parameter list as the actual arguments. That is, the first actual argument **x** is a float and so in the first formal argument **base**. Similarly the second actual argument **y** is an integer and so is the second formal argument **index**. *The number, type and order of formal arguments must match the number, type and order of the actual arguments for the function to function properly*. In this case the **float** value **1.5** is passed to the `float` variable **base** and the **int** value **4** is passed to the `int` variable **index**. The statements within the function are placed within a pair of curly brackets **{}** just as in **main()**. In the above example, the result of $base^{index}$ is stored in the **float** variable **result**. The ***for*** loop inside the function calculates the power. The loop counter **i** is initialised to 1 and is used to count the number of times the `base` is to be

multiplied i.e. it goes upto a value given by the `index`. After the calculations, this value is passed to the calling function by the keyword **`return`**, which passes the program control back to the point from where the function was called. Thus the statement **`return result;`** sends the *value* of the calculated float variable **`result`** i.e. **`5.062500`** to the float variable **`output`** in the **`main()`** function. The data type **`float`** written *before* the name of the function indicates that the data type of the return value is a float. The return value is then assigned to the variable **`output`**. One thing should be ensured that ***the data type of*** **`output`** ***should be the same as the data type of the return value***.

Note1: The variable **`result`** is local to the function power i.e. the variable is created each time the function is called and *ceases to exist* after the function has completed its execution. The question that may rise is that if the variable ceases to exist then how is the value transferred to the calling function. The answer lies in the fact that a *copy* of the value to be returned is made, and this copy is made available to the return point in the program.

Note2: Similarly when a function passes a value to the called function, it makes ***copies*** of the values of the argument variables and supplies these *copies* to the called function. The values of the ***actual arguments*** thus remain ***intact*** even if the *formal argument* values are changed by the called function.

Note3: Though any function can be called from any other function, but a function *cannot be defined* inside another function. Thus in this case **`power()`** cannot be defined inside **`main()`**. But the order in which the functions are defined *may not be the same* as the order in which they are called.

5.4 Declaring a Function prototype:

A prototype for the function provides the compiler with the basic information about how the function is used. It indicates the **function name**, the **function argument data types and names**, and the **function return data type**. The prototype declaration has the *same* syntax as the function definition header with the exception that a **semicolon** is added at the end of the header to distinguish it from a function definition. We write below the previous program by using a function prototype. Note that since a prototype has been declared, it becomes unnecessary to define the function before it is been called by **`main()`** and hence is generally defined after the closing braces of the **`main()`** function.

```
#include<stdio.h>
float power(float base, int index);

main()
{ float x, output;
  int y;
  printf("\nProgram to Calculate x^y");
  printf("\nEnter base x (decimal value)");
  scanf("%f", &x);
  printf("\nEnter index y (integer value)");
  scanf("%d", &y);
  output=power(x,y);
  printf("\nValue of x^y is %f", output);
  return(0);
}

float power(float base, int index)
{ float result=1;
  int i=1;
  for(;i<=index;i++)
       result*=base;
  return result;
}
```

Semicolon (`;`) after the ending bracket differentiates this from a function definition.

Function prototype declared before the `main()` function

Function name within another function (in this case `main()`) identifies this as a function call.

No semicolon (`;`) after the ending bracket differentiates this from a function prototype. During definition there should not be any semicolon after the function name.

Variables used by the function `power()` declared within `power()`.

As is evident from the above program piece, the function prototype for the user-defined function **`power()`** i.e. **`float power(float base, int index);`** indicates the following:

- The **return data type** of the function **`power()`** is a **`float`** i.e. the function returns a **`float`** value
- The **name** of the function is **`power`**
- The **argument list** consists of a **`float`** type data variable with name **`base`** and an **`int`** type data variable with name **`index`**

After the declaration of the function prototype, when the function is being called in the **`main()`** function, the two values **`x`** and **`y`** are passed to **`base`** and **`index`** and the calculations are done as before and finally

the result returned to the **main()** function. When using *several* functions, the function prototypes of all the functions are stated before **main()** and then defined one by one generally after the **main()** function.

5.5 The return statement:

As was mentioned earlier, the return statement is used to *send back* control to the calling function. It also returns any value (can be result of any calculation) that is placed after the keyword return. The value may be any **literal** or can be any **variable** name or an **expression**, that evaluates to a certain value (decimal or logical) or any character. Thus the following are all valid return statements:

Ex2:

```
int bonus(int x)                    → Function called bonus, receives an int and returns an int
{ printf("\nScore =%d", x*x);       → Function body prints the score
  return (100);                     → Returns the constant literal int value 100
}
```

Ex3:

```
double molecules(int moles)   → Function called molecules, receives an int and returns a double
{ printf("\nThe total number of moles of gas=%d", moles);
  return (6.023e23*moles);    → Returns the result of the double value (6.023e23*moles)
}
```

Ex4:

```
float tot_press(float p1, float p2)      → Function called tot_press returns a float
{ printf("\nThe partial pressures are %f and %f", p1, p2);
  return (p1+p2);                   → Returns the result of the float value expression (p1+p2)
}
```

Ex5:

```
int complement(int x)       → Function called complement, receives and returns an int value
{ printf("\nMistakes can fetch you only lesser points!!");
  return (!x);              → Returns the result of the logical operation !x
}
```

Ex6:

```
float volume(float p, float t)    → Function called volume, returns a float
{ float vol;                → Variable called vol of type float declared within the function
  vol=8.1*t/p;              → Function calculates and stores result in vol
  return vol;               → Returns the result stored in the float variable vol
}
```

Ex7:

```
char yes()                          → Function called yes when called returns a character
{ printf("\nThe answer is correct");
  return ('Y');                     → Returns the character literal Y
}
```

Ex8:

```
char input_section()                → Function called yes_no, returns a character
{ printf("\nEnter the Section in which you study");
  return (getchar());               → Returns the character input via getchar( )
}
```

Ex9:

```
int max_value(int x, int y)   → Function called max_value, returns the maximum of two values
{ printf("\nThe maximum of two values calculated");
  return ((x>y)?(x):(y));     → Returns the greater of x and y by using the conditional operator
}
```

In all the above cases the function returned certain values to the calling function, be it an int, a float, a char or a double. As mentioned earlier, the type of data returned by the function is mentioned at the beginning of the function definition / declaration. In case no such thing is mentioned, then ***the default data type returned by a function is always an int***. Thus in case a function returns a float value and the return data type is not mentioned before the function name, then correct data *will not* be transferred to the calling function.

Ex10: The following example is used to find the cube of a float variable. The variable to cube is passed to the function called **cube()** which does the cubing and sends back the result via the return statement.

```
#include<stdio.h>
cube(float x);                          → Function prototype for cube() declared

main()
{ float x, output;
  printf("\nEnter the decimal number to cube"); scanf("%f", &x);
  output=cube(x);
  printf("\nThe cube of %f is %f", x, output);
  return(0);
}

cube(float x)                           → Function called cube, receives the float value x
{  return (x*x*x); }                    → Returns the cube of x
```

In the above example if someone inputs the value of say **x=5** then the return result will be 125 as expected. But if someone enters 5.5 then instead of getting 166.375 one will get the result as 166. The reason is that, return type of the function being not declared explicitly, it is taken by default as an **int.** Hence the function returns only the integer portion of the result. To rectify the problem, the function cube is to be declared as a float as shown below:

```
float cube(float x)                     → Function called cube declared as a float
{  return (x*x*x); }                    → Returns the cube of x as a float
```

In all the above examples, each function had only one return statement. In general a function can contain as many return statements as required but ***can return only one value at a time to the calling function***. Moreover it is not necessary that the return statement should be placed at the end of the function; it *can be placed anywhere* in the function.

Ex11: The following program inputs a character and if it is a lower case character then it converts it to upper case else it prints out the original character.

```
#include<stdio.h>
char convert_to_upper(char initial);    → Function prototype for convert_to_upper

main()
{ char lower, upper;
  printf("\nEnter any character"); scanf("%c", &lower);
  upper=convert_to_upper(lower);        → The character in lower sent to the called function
  printf("\nThe upper case letter is %c", upper);
  return(0);
}

char convert_to_upper(char letter)      → Function receives the character in letter
{
  if(letter >='a' && letter <= 'z')
    return ((letter-'a')+'A');          → Returns upper case letter corresponding to letter
  else
    return (letter);                    → Returns the letter, if already upper case letter
}
```

The function **convert_to_upper()** uses the ASCII values of the input character **letter** to check if it is lower or upper case. If found to be of lower case, (**letter >='a' && letter <= 'z')**, then it converts it to upper case by the expression ((**letter-'a')+'A')** and returns that value. Else it is already an upper case letter and hence returns the unchanged character via the second return statement.

Suppose a person has entered the letter ‘c’. The ASCII values of the required characters are:

A=65, Z=90, a=97, z=122, c=99, C=67.

Thus c=99 is >a=97 and <z=122 and hence satisfies the ***if*** condition. The value returned is thus equal to:

((letter-'a')+'A')= ((99-97)+65) = (2+65) = 67 = ASCII of C.

In general, whenever a program branching is there and the output depends on which program segment the program has branched to, then the different branches can each contain a return statement to return a

particular value. For example a function may contain a **switch-case-default** statement, where each **case** can return a particular value.

It is not necessary that a function should always return a value. A function may be used to print some messages or carry out a calculation and print the result without passing any value to the called function. Under such cases, to make the compiler understand that the given function does not send back any value, the keyword **void** is placed before the function name in place of the return data type. In such a situation, no return statement should be used in the function; otherwise it will give error message during compilation. We modify the maximum value finding function (**Ex.9**) to shows the use of the keyword **void**.

```
#include<stdio.h>
void max_value(int x, int y);      → Function prototype for max_value() declared as void

void main(void)                    → main() defined as returning void and receiving void
{ int a, b;
  printf("\nEnter number1: "); scanf("%d", &a);
  printf("\nEnter number2: "); scanf("%d", &b);
  max_value(a,b);                  → No return(0) statement after this as function has been
}                                    declared to return nothing i.e. void

void max_value(int x, int y)       → Function called max_value, returns nothing
{
  printf("\nThe maximum of the two numbers = %d",((x>y)?(x):(y)));
}                                  → No return statement after this, as in case of main()
```

In the above example the function **max_val()** has been defined to return nothing i.e. **void**. Similarly the **main()** function has also been defined as **void** as it is not returning any value to any other function. Moreover as it is not receiving any value either i.e. any formal argument, the keyword **void** has been placed within the brackets after **main** to indicate the ***absence of any argument***. A function defined as **void** should not contain any **return** statement and hence the usual **return(0)** statement is absent.

5.6 Different uses and ways of accessing a function:

In C, a function can be defined to perform a variety of jobs. It can be used to carry out some calculations, print out some results or simply print some heading. Dividing a program into a number of functions usually reduces the chances of an error and helps in program understanding and debugging. So use of as many functions as possible is a good programming practice. As said earlier, as a function can be used for just any type of job, let us now discuss the different types of uses of functions.

Ex12: The following function is used just to type the heading of a program.

```
void heading(void)                 → Function receives nothing, returns nothing
{
   clrscr();
   putchar("\n");
   printf("\n      ******************************************");
   printf("\n      *       PROGRAM TO CALCULATE FACTORIALS     *");
   printf("\n      ******************************************");
   putchar("\n");
}
```

The function named **heading** is neither receiving any values as formal arguments from the calling function, nor is it returning any values to the calling function. Accordingly it has been declared as **void**. Moreover since it is also receiving nothing as formal arguments, the keyword **void** is written within the brackets to indicate the same.

Ex13: The function given below is our old prompt. It receives nothing as formal arguments.

```
char prompt(void)                  → Function receives nothing, returns a character
{ puts("\nPress R to repeat, E to exit");
  return (getchar());              → Function returns the character input by the user
}
```

Ex14: The following function receives the float variable centigrade, converts it to fahrenheit and prints the result without returning anything. It uses a *local variable* called `f` to calculate the result.

```
void fahrenheit(float centigrade)        → Function receives float, returns nothing
{
  float f;                               → Local variable f declared as float within the function
  f=(9.0/5.0)*centigrade+32;
  printf("\n%f degree Centigrade = %f degree Fahrenheit", centigrade, f);
}
```

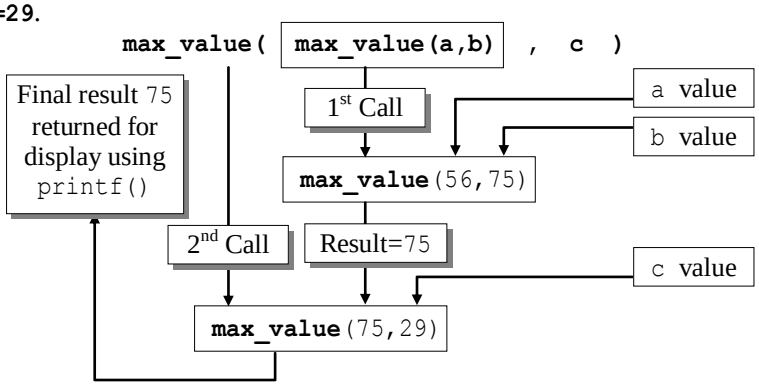
Ex15: The following program shows how the same function discussed in **Ex.9,** to calculate the maximum of two integers, can be used to find the maximum of three integers.

```
#include<stdio.h>
int max_value(int x, int y);

void main(void)
{ int a, b, c, d;
  printf("\nEnter number1: "); scanf("%d", &a);
  printf("\nEnter number2: "); scanf("%d", &b);
  printf("\nEnter number3: "); scanf("%d", &c);
  printf("\nThe maximum number is %d", max_value(max_value(a,b),c));
}

int max_value(int x, int y)          → Function called max_value, returns the max of x and y
{ return ((x>y)?x:y);
}
```

In the above example, the function **max_value()** has been used *both* as an argument to the **printf()** function and as an argument to a call of the same function. It shows the practical use of a general function that is called several times to execute different jobs using the same function and hence making the code more compact. Thus instead of writing different functions to find the maximum of two and three numbers, we have used the **max_value()** function twice to achieve our job. The situation can be thought of as nesting of two functions. The diagram below shows the order in which the functions get evaluated for **a=56**, **b=75**, **c=29**.



The inner **max_value()** function works on **56** and **75** and returns **75** as the maximum of two values to the outer **max_value()** function which now has the arguments **75** and **29** and returns **75** as the final maximum value after the comparison. In this way by nesting several **max_value()** functions we can find the maximum of several integers by taking two integers at a time and using the **max_value()** function for two integers at a time.

Ex16: The following program uses a function in a loop to calculate and print a power table.

```
#include<stdio.h>
double power(double x, int y);

void main(void)
{ double base; int i, index;
  printf("\nEnter base (decimal value): "); scanf("%lf", &base);
  printf("\nEnter index (integer value): "); scanf("%d", &index);
  for(i=0;i<=index;i++)
       printf("%8.0f", power(base,i));
}
double power(double x, int y)        → Function called power
{
  double result=1.0;
```

```
  if(y<0)                          → The condition y<0 checks for a –ve input to prevent infinite
       return 0.0;                   looping by returning 0.0, not entering the while loop
  while(y--)                       → The condition y-- counts the number of times to repeat
       result*=x;                    and terminates when y becomes equal to 0.
  return result;
}
```

When executed with base=2 and index=8, the program prints out the following:

1 2 4 8 16 32 64 128 256

With **base=2** and **index=8**, the loop starts by passing the values **(2,0)** to **power()**. Inside the function **power()**, the while statement is used to calculate the required result. For **y=0**, the while condition evaluates to false [as the postfix operator **y--** decrements the value after the execution of the condition] and **returns** the value of **result** as 1.0 i.e. the initialised value. For any subsequent value of **i**, say **3**, the **for** loop passes the values **(2,3)** to **power()**. The **while** statement within **power()** will execute **3** times with each time the value of **y** decreased by **1** until the **y--** makes **y=0**, when the **while** loop will stop and **return** the value stored in **result**.

Ex17: The following program shows that a **function** when called (by value) **does not change the values of the actual arguments**.

```
#include<stdio.h>
void exchange(int x, int y);

void main(void)
{ int num1, num2;
  printf("\nEnter number1: "); scanf("%d", &num1);
  printf("\nEnter number2: "); scanf("%d", &num2);
  printf("\nThe initial value of number1=%d", num1);
  printf("\nThe initial value of number2=%d", num2);
  exchange(a,b);
  printf("\nThe final value of number1=%d", num1);
  printf("\nThe final value of number2=%d", num2);
}

void exchange(int num1, int num2)
{
  int temp;
  temp=num1;                       → The value of num1 temporarily stored in the variable temp
  num1=num2;                       → The value of num2 assigned to num1
  num2=temp;                       → The value of temp i.e. of old num1 assigned to num2
  printf("\nThe new value of number1=%d",num1);
  printf("\nThe new value of number2=%d",num2);
}
```

When the above program is run and number1 is entered as 25 and number2 as 52 say then the output will be:

```
The initial value of number1=25
The initial value of number2=52
```
} Initial values of **num1** and **num2** in **main()**

```
The new value of number1=52
The new value of number2=25
```
} Changed values of **num1** and **num2** in **exchange()**

```
The final value of number1=25
The final value of number2=52
```
} Original values of **num1** and **num2** in **main()** remain same

As discussed earlier, while passing the values to the called function, C makes ***copies*** of the variables' values and sends the copies to the called function. Thus the **actual values** of the arguments remain **intact**. This mode of passing variable as arguments is known as **pass by value method**. Later while discussing pointers, we will learn another method called **pass by reference method**, where instead of making a copy of the variable value and sending this copy to the called function, the address of the memory location of the variable is passed. By this method the actual value of the argument can be changed in the called function.

5.7 Recursion:

In C a function can be made to **call itself** and a function that has been constructed in such a manner as to call itself is known as a recursive function. This is sometimes also called circular definition i.e. the process of defining something in terms of itself. For a function to be recursive, it must follow two basic rules:

1. **It must have an ending point**
2. **It must make the problem simpler**

The simplest function to illustrate the principles of a recursive call is the function to calculate the factorial of a number. We had earlier calculated the factorial of a number by using the ***for*** loop. Now we will carry out the same function without using the ***for*** loop but using function recursion.

By definition a factorial is defined as:

- **`factorial(0)=1`**
- **`factorial(n)=factorial(n-1)`**

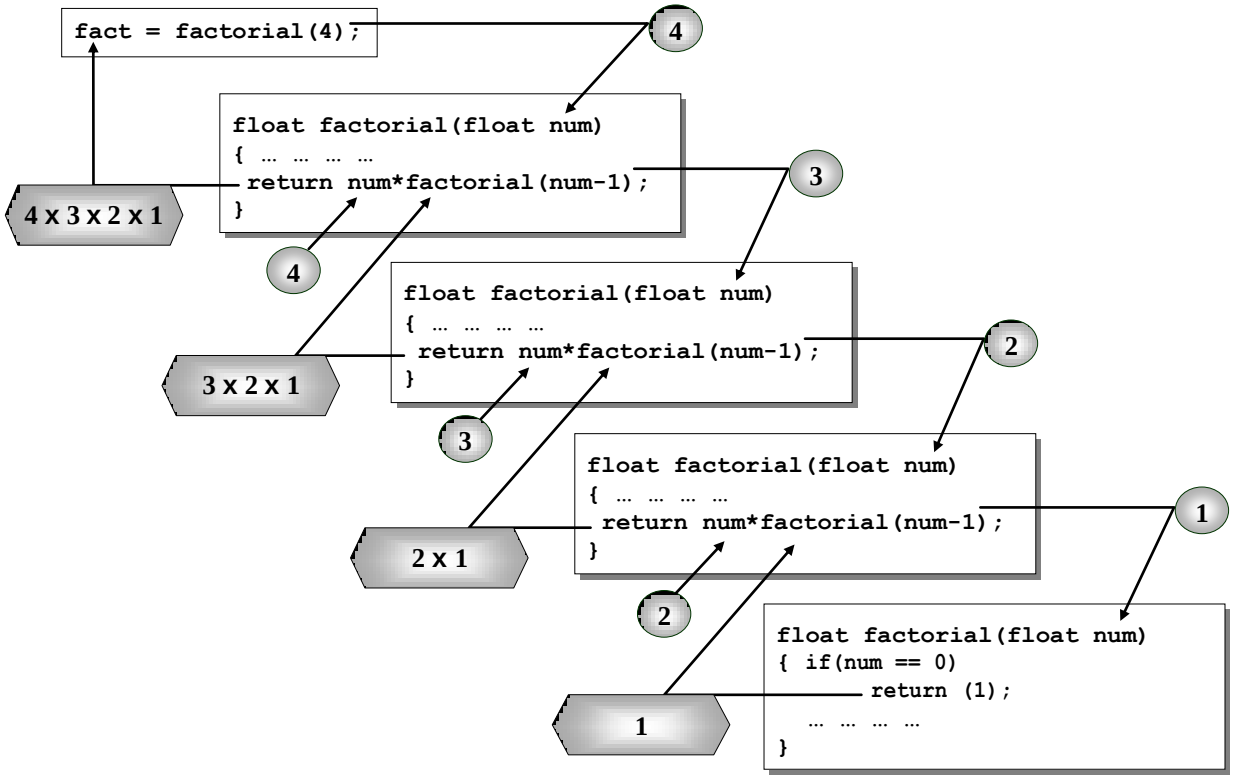
The above two rules of a factorial satisfy the function rules for recursion i.e. the first definition of the factorial serves as an ending point while the second definition simplifies the problem as the calculation of factorial(n-1) is simpler than factorial(n). Armed with the above rules and definitions le us now write down the recursive version of the factorial calculation function.

```
#include<stdio.h>
float factorial(float num);

void main(void)
{ float num, fact;
  printf("\nEnter number to find factorial: "); scanf("%f", &num);
  fact=factorial(num);
  printf("\nThe factorial of %.0f is = %.0f", num, fact);
}

float factorial(float num)
{ if(num==0)                          → Condition to stop the recursion
        return (1);
  else
        return num*factorial(num-1);  → Recursive call to the function made by this statement
}
```

Let us now find how the function operates by calculating the factorial of say 4:



The above diagram illustrates the flow of program control for calculating the factorial of 4. The main function calls **factorial()** with 4 as its actual argument and the function in return returns back the calculated value for factorial 4. But before returning the computed value, the function calls itself with the actual argument as (4-1) = 3 and expects a return value and waits. This second call to the function with 3 as the actual argument causes another call to the function with (3-1) = 2 as the actual argument. This again leads to another self function call with actual argument as (2-1) = 1. At this point, the called function has a formal argument of ‘1’ which satisfies the ***if*** condition and hence returns ‘1’ without executing any further function calls. In this way the return chain continues until all the calls are returned.

Ex18: The next program is used to calculate the value of x^y using recursion.

```
#include<stdio.h>
float power(float num);

void main(void)
{ float base, output;
  int index;
  printf("\nEnter base: "); scanf("%f", &base);
  printf("\nEnter integer index: "); scanf("%d", &index);
  output=power(base,index);
  printf("\nThe result of x^y is = %f", output);
}

float power(float x, int y)
{ if(y==0)                          → Condition to stop the recursion
    return (1);
  else
    return x*power(x,(y-1));        → Recursive call to the function made by this statement
}
```

the above function works in a similar way as the last function. The *terminating condition* for the loop is when the index becomes equal to ‘**0**’ i.e. for **power(x,0)**, as $x^0=1$. For all other values of the index y, the return value of the function is the product of the base times the previous power of index. The calculation of 2^3 shown below clarifies the working of the function.

power(2,3)=2•power(2,2)=2•2•power(2,1)=2•2•2•power(2,0)=2•2•2•1=8 [as `power(2,0)=1`]

Ex19: The previous programs used recursive functions involving products. Let us now see how a recursive function can also be used to do additions. The following program finds the sum of an A.P. using recursion.

```
#include<stdio.h>
int ap_term(int b, int t);
int main(void)

{ int i, ft, cd, num, sum=0;
  printf("\nEnter first term of A.P.: "); scanf("%d", &ft);
  printf("\nEnter common difference of A.P.: "); scanf("%d", &cd);
  printf("\nEnter number of terms to sum: "); scanf("%d", &num);
  if (num>0)                        → Condition prevents function call with negative num
     for(i=0; i<num; i++)           → Each term of the AP series is added by the for loop
       sum=sum+(ft+ap_term(cd, i) );  → Function ap_term() is called to generate each term
  printf("\nThe sum of the given A.P. = %d", sum);
}

int ap_term(int b, int t)           → Value of i is received by the formal argument t
{ if (t==0)                         → Condition to stop the recursion when t=0
     return (0);
  else
     return b + ap_term(b,(t-1)); }  → Recursive call to function made by this statement
```

The function **ap_term()** is used recursively to find the **i**th term of the A.P. and is called with the arguments **cd** as the common difference i.e. the term that will go on adding and **i** i.e. the number of common differences to add for the **i**th term. The first term **ft** is added to the **sum** during the first call to the function with i=0. Within the function, the **return** value is equal to the sum of the common difference plus the call to the same function with number of terms one less. This continues till the number of common differences to add for the **i**th term becomes equal to 0, which serves as the limiting condition and returns the last value to add as 0. The sum of the A.P. is obtained in the variable **sum**, by adding each of the terms.

Ex20: The following program uses recursion to reverse the order of input of a character string.

```
#include<stdio.h>
void reverse(void);

void main(void)
{
  printf("\nEnter a line of text:\n");
  reverse();
}

void reverse(void)
{ char c;
      if((c=getchar())!='\n')      → Condition to stop the recursion with '\n' entered
            reverse();
      putchar(c);
  return;
}
```

The above simple program uses recursion to reverse the characters entered in a string. The recursive function reverse(), simply reads a single character at a time until an end of line character (\n) is reached. This is used as the condition to stop the recursion. When a recursive function is called, the successive data items are placed on a stack, which is a *last-in-first-out memory structure*, where each successive data items are pushed down upon the preceding data items. When the condition that terminates the recursion is encountered, the function calls are been executed in the reverse order. Thus the last in item is pushed out first followed by the second last and so on and so forth till the stack becomes empty.

In our case, when an end of line character in encountered, the recursion stops and the last data item entered is first called from the stack and displayed by the putchar(c) statement. The preceding characters then follow out of the stack in the reverse order and gets printed onto the screen. A sample run of the program is shown below:

```
Enter a line of text:

Time to Quit              → String input by the user

tiuQ ot emiT              → Reversed string output by the recursive function
```

Though recursion is a useful tool, as it can sometimes make a program much more compact, but before going for recursion one should also check for other alternatives also such as using a loop. This is because a program execution using loops can at times be efficient in reducing overhead compared to recursive calls. This is because at each recursive call, the compiler may have to generate copies of the arguments to the function and keep track of the location to return to when each return statement is executed, adding to the overhead.

5.8 Storage Classes:

In C every variable has two attributes. They are the **scope** and the **class** of the variable. By scope of a variable we mean the *area of the program* in which the variable is *valid* and by class of a variable we mean whether the variable is *temporary* or *permanent*. Till the point we dealt with functions, we used variables only within the **`main()`** function block. But the moment we defined other functions besides the **`main()`** function, we had variables defined both in the **`main()`** function and in the user-defined functions. Under such circumstances a variable can have several scopes or options i.e. it can be valid only within **`main()`** or only within the user-defined function or can be valid in both the functions. Moreover if the storage class of the variable is not mentioned in the declaration, then the compiler assigns the *default* storage classes based on the context in which the variable was used. Hence in all the previous programs, the storage classes were assigned the default values. Each variable occupies a portion of the memory to store its value. The variable's storage class determines in which location of the computer, the values in the variables are stored i.e. whether in the **Memory** or in the **CPU Registers** (which are special memories located within the CPU).

Based on the scope and class of the variables, they are divided into 4 different types. These are:

1. **Automatic** or local storage class variables
2. **Register** storage class variables
3. **Static** storage class variables
4. **External** or Global storage class variables

The table below gives a comparative study of the scope and class of the above four storage classes:

Storage Class	Storage	Default Initial Value	Scope	Life
Automatic (syntax: **auto**) (All data types)	Memory	Unpredictable Garbage value	**Local** to the block in which the variable is defined	Till the control remains within the block in which it is defined
Register (syntax: **register**) (Only integers)	CPU Registers	Unpredictable Garbage value	**Local** to the block in which the variable is defined	Till the control remains within the block in which it is defined
Static (syntax: **static**) (All data types)	Memory	Zero	**Local** to the block in which the variable is defined	Value of the variable persists between different function calls
External (syntax: **extern**) (All data types)	Memory	Zero	**Global**	As long as the program's execution does not come to an end

Automatic Storage Class variables can be defined at the beginning of any block of code immediately after the opening brace (i.e. after the **{**). They are said to have local or block scopes. This means that they are **defined only within the particular block** (defined by a pair of curly brackets **{ }**) **in which they are declared**. They are born at the time the program control enters that block and ceases to exist when the control comes out of the block. Accordingly the same variable name can be used in more than one block with no fear of the compiler mixing up which variable to use when. This is because the current value of an automatic variable is the value of that particular variable in the current block. The following function block demonstrates the use and scope of the automatic class. Note that when not initialised, the value of the variable displayed will be garbage and within each block the variable has its own value.

```
main()
{  auto int count1, count2=20;
   printf("\nOuter count1 (uninitialised)=%d", count1);
   count1=10;
   printf("\nOuter count1 (initialised)=%d", count1);
   printf("\nOuter count2 (initialised)=%d", count2);

   {    auto int count1=30, count3=40;
        printf("\n****************");
        printf("\nInner count1=%d", count1);
        printf("\n****************");
        count1 += 50;
        count2 += count3;
   }

   printf("\nOuter count1=%d", count1);
   printf("\nOuter count2=%d", count2);
   return (0);
}
```

Inner Block

Outer Block

The **output** of the above program will give the following:

```
Outer count1 (uninitialised)=575        → Un-initialised garbage value of count1
Outer count1 (initialised)=10
Outer count2 (initialised)=20
****************
Inner count1=30                         → Value of count1 as initialised in the inner block
****************
Outer count1=10                         → Outer block count1 contains same old value
Outer count2=60
```

The variables **count1** and **count2** are declared in the outer block and initialised. When the local variable **count1** is again declared in the inner block, it ***hides*** the original **count1**. The inner **count1** ceases to exist the moment the control comes out of the inner block. Hence the next **printf()** displays the same initialised value of **count1** as 10, as was declared in the outer block unaffected by the **count1+=50** statement. However **count2**, which was declared in the outer block has changed in value by using a variable in the inner block. This was possible because, since the outer block encompasses the inner block, the variable declared at the beginning of the outer block is available in the inner block (unless a variable of the same name is declared in the inner block as with **count1**). Hence the value of **count2** is displayed as **60** i.e. **20+40**.

In case no class specifier is mentioned, the variable is assumed to be automatic, making the above declaration optional. The next example shows the use of automatic variables by functions.

```
#include<stdio.h>
void multiplication_table(int x);
void main(void)
{ int i;                              → Variable i declared in main()
  printf("\n          Multiplication Table from 2 to 12\n\n");
  for(i=1;i<=12;i++)
       multiplication_table(i); }

void multiplication_table(int x)
{ int i;                              → Same variable name used for the function loop counter as i
  for(i=1;i<=12;i++)
       printf("%4d",x*i);
  putchar('\n'); }
```

The above program prints out our good old multiplication table. Note that the same variable **i** has been used both in **main()** and in the function **multiplication_table()** without any conflict. This was possible as the variables are automatic in nature. With each function call, the value of **i** in the **main()** function ***for*** loop is passed to the called function, where another variable **i** is declared whose scope exists only within that function body and hence causes no conflict.

Register Storage Class variables are similar to auto variables with the only exception that they generally occupy the CPU register memory spaces instead of the conventional memory. Operations using register variables are faster in comparison to conventional memory stored variables. But not all variables can be defined as register variables as the allowed data types for these are **only integers** (some compilers may also support character type variables). Hence program loop counters and similar variables which are accessed frequently are declared as register variables. But care should be taken while choosing register variables as only 16 registers are allowed in a PC. Moreover not all registers are available at all times for storing these variables. This however will not cause any problem in program running as, in case register memory is not available then the register variable is automatically changed to an automatic variable by C. The syntax is:

```
register int index;
```

Static Storage Class variables have also scope within the block in which they are declared, but they are initialised only once and their default initial value is ‘0’. Whereas for the previous types of variable classes the value stored in a variable is lost forever once the control comes out of the block in which it is defined, in case of static variables, once initialised, the value persists between different function calls. This type of use can be particularly useful when the contents of a variable need to be accumulated between different function calls such as to count how many times a function has been called as shown below:

```
#include<stdio.h>
void counting(void);
int main()
{ int i;
  counting();
       for(i=0;i<=2;i++)
           counting(); }

void counting(void)
{ static int count;
  printf("\nFunction counting() called %2d times", ++count); }
```

In the above example, the function **counting()** is first called by **main()** where it need not be initialised, as the default initial value of a static variable is ‘0’. This results in printing of the first line as shown below. Before printing, the variable count is increased by the pre increment expression ++count. The next calls to the function through the ***for*** loop in **main()** result in increment and printing of the value of count 3 times (**i=0 to 2**) resulting in the output shown below.

```
Function counting() called 1 times
Function counting() called 2 times
Function counting() called 3 times
Function counting() called 4 times
```

In place of static variables if the **count** in the function **counting()** was declared as an automatic variable and initialised to 0, then the value of count would not persist between function calls and each time the function is called it will get initialised to 0. Thus the resulting output would have been:

```
Function counting() called 1 times
Function counting() called 1 times
Function counting() called 1 times
Function counting() called 1 times
```

As a general note of caution, **avoid using static variables unless one really needs them**. Since their values are kept in memory even if they are not used and they are alive throughout the life of the program, they may use up a lot of memory space unnecessarily.

External Storage Class variables differ from the variables we have discussed so far mainly with respect to their scope. They have a **global scope** i.e. they are not defined or available within a particular block of code but are available throughout the program. As such they are also defined **outside** all functions, including **main()**. External variables are usually defined before **main()**, just like normal variable declaration.

As external variables are global in nature, they can be assigned a value in one function and the same value can be used in another function. We know that a function in general can return a single value. However using global variables, more than one value calculated in a function can be made available to the calling function. Moreover **arrays** declared as global variables can be passed between functions easily. Thus when a variable is used by many functions in a program, it can be declared as an external variable, thus avoiding the need to pass the variable between functions. The following example shows the use of external variables:

```
#include <stdio.h>
#include <math.h>
void real_roots(float a, float b, float c);      → Function real_roots() declared

float x1, x2;                                    → The external variables x1 and x2 defined
void main()                                        outside the main() function
{ float a, b, c;
  printf("\nEnter the coefficients of the quadratic equation ax^2+bx+c=0: ");
  scanf("%f %f %f", &a, &b, &c);
  real_roots(a, b, c);                           → Function real_roots() called
  printf("\nRoots of the quadratic equation are: x1=%.3f, x2=%.3f", x1, x2); }

void real_roots(float a, float b, float c)
{ float m, n;
  m = -b/(2*a);
  if( (b*b-4*a*c)<0 )                            → Condition checked for valid real roots
    { printf("\Equation has no real roots");
      exit(0); }                                 → Program is exit if the roots are not real
  n = sqrt(b*b-4*a*c)/(2*a);
  x1 = m+n;                                      → Roots are calculated into the global
  x2 = m-n; }                                      variables x1 and x2.
```

Output of the above program when run:

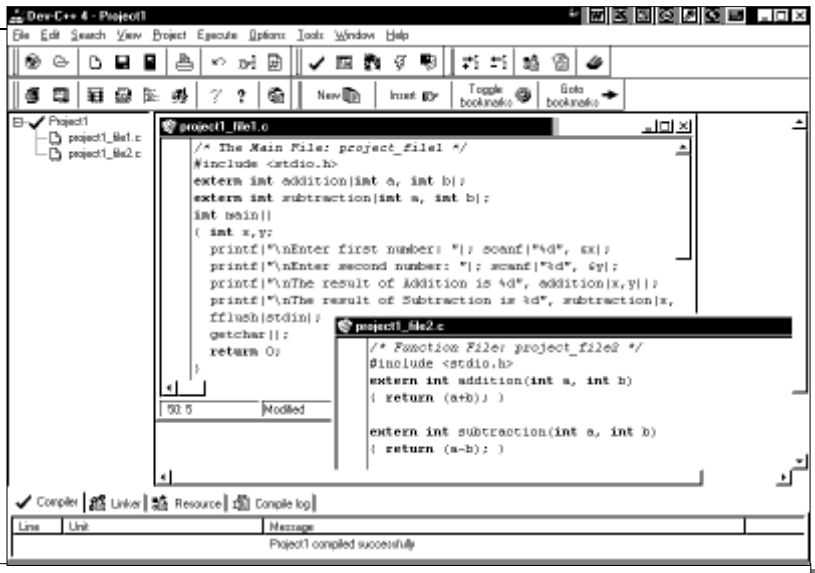
```
Enter the coefficients of the quadratic equation ax^2+bx+c=0: 1 1 -6
Roots of the quadratic equation are: x1=2.000, x2=-3.000
```

The value of the roots are calculated within the function **real_roots()** and assigned to the variables **x1** and **x2**. Since the variables are declared as global variables, they need not be returned and the values assigned to them in **real_roots()** are also available in the **main()** function, where they are printed.

5.9 Multi-file Programs:

A ‘C’ program can consist of multiple files, with each file containing the code for a specific task like a lengthy function or a collection of user defined functions engaged in similar type of work. A user can write frequently used user-defined functions and store them in separate files. When required, these files can be linked with other programs and a complete program can be obtained, thus eliminating the need to write the functions every time they are needed in a program. These individual files can then be compiled separately and then linked together with the main program to form a complete executable program. However when dealing with multi-file programs, the **scope of functions and variables** need to be carefully outlined. Within a multi-file program, depending upon its scope, a function or a variable can also be either static or external. When defined as static, it is valid within the file in which it is declared, while if defined as external, it has a scope throughout the program.

When using functions in multi-file programs, they need to be **defined first in any one of the files** as an external type function by using the keyword **extern** before the function header. In this regard note that a function can be defined in one file only. However, when a function is defined in one file and accessed in another, then the second file must include a function declaration, indicating that the function’s definition



appears elsewhere, just like functions are declared in a single file program. By default, a function is assumed to be of external type if no storage class is mentioned.

To execute a multifile program, the source files are placed within a project file. Each individual file is then compiled and the resulting object files are linked together to form an executable program. The following example illustrates the use of multifile programs by using functions called **add()** and **subtract()** in one file and accessing these functions from another file in a project. The project environment for the said project is shown above.

```
/* The Main File: project_file1 */
#include <stdio.h>
extern int add(int a, int b);          → Function declared as external and used in this file
extern int subtract(int a, int b);     → Function declared as external and used in this file
int main()
{ int x,y;                             → Local variables declared
  printf("\nEnter first number: "); scanf("%d", &x);
  printf("\nEnter second number: "); scanf("%d", &y);
  printf("\nThe result of Addition is: %d", add(x,y));          → Function called
  printf("\nThe result of Subtraction is: %d", subtract(x,y)); → Function called
  return 0;
}
```

```
/* User Defined Function File: project_file2 */
#include <stdio.h>
extern int add(int a, int b)           → Function defined in the second file as external
{ return (a+b); }

extern int subtract(int a, int b)      → Function defined in the second file as external
{ return (a-b); }
```

Output of the above program, when compiled and run:

```
Enter first number: 12
Enter second number: 5
The result of Addition is: 17
The result of Subtraction is: 7
```

Similar to functions, **external type variables** can also be **defined** in one file and then accessed from another file. To do so, they are first defined and initialised (optional) in one file and used in another file by **declaring them as `extern`** type variables outside all function definitions in that file.

5.10 Arguments of the main( ) function:

Just like any other function, the **`main()`** function can also have its own set of arguments. These arguments allow parameters to be passed from the operating system to the program, through the **`main()`** function. In general two such arguments are allowed, viz. **`argc`** and **`argv`**. The first one i.e. **`argc`** is an integer type variable and indicates the number of parameters passed to the main function. The second one i.e. **`argv`** is an array of strings (i.e. array of pointers to characters) with each string representing a parameter that is passed to **`main()`**. More of these will be discussed in later chapters dealing with file handling operations. We give below only the **general format** of the arguments of the **`main()`** function:

```
int main(int argc, char *argv[])
```