1. Functions

1.1 Introduction to Functions

Every C program must have a **main** function to indicate where the program has to begin its execution. If a program is written only using a single **main** function, the program becomes too large and complex and as a result, the task of debugging, testing and maintaining becomes difficult.

C functions can be classified into two categories:

- **Library Functions:** Not required to be writtenby the programmer. e.g. `printf()`, `scanf()`, `strlen()` etc.
- **User-defined functions:** Has to be developed by the user at the time of writing a program.

If a program is divided into functional parts, then each part may be coded independently and later combined into a single unit. These independently coded programs are called sub programs that are much easier to understand, debug and test. In C, such subprograms are referred to as **functions**.

The advantages of using functions are:

1. Uses **top down modular programming** approach. The high level logic of the overall problem is solved first while the details of each lower functions are taken care later.

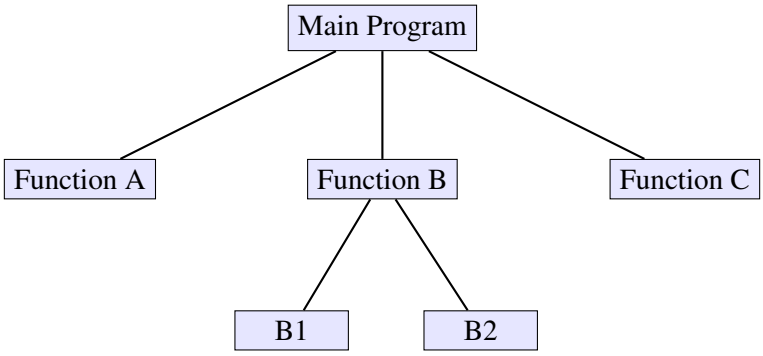

Figure 1.1: Top-down modular programming using functions

2. The length of the program can be reduced by using functions at appropriate places.
3. It is easier to locate and isolate faulty functions.
4. A function once created, can be used by many other programs.

A function is a self contained block of code that performs a particular task. Once a function has been designed and packed, it can be treated as a unit that takes some data from the main program and returns a value. The inner details of the operations are invisible to the rest of the program. All that the program knows about a function is what goes in and what comes out. Every C program can be designed using a collection of functions.

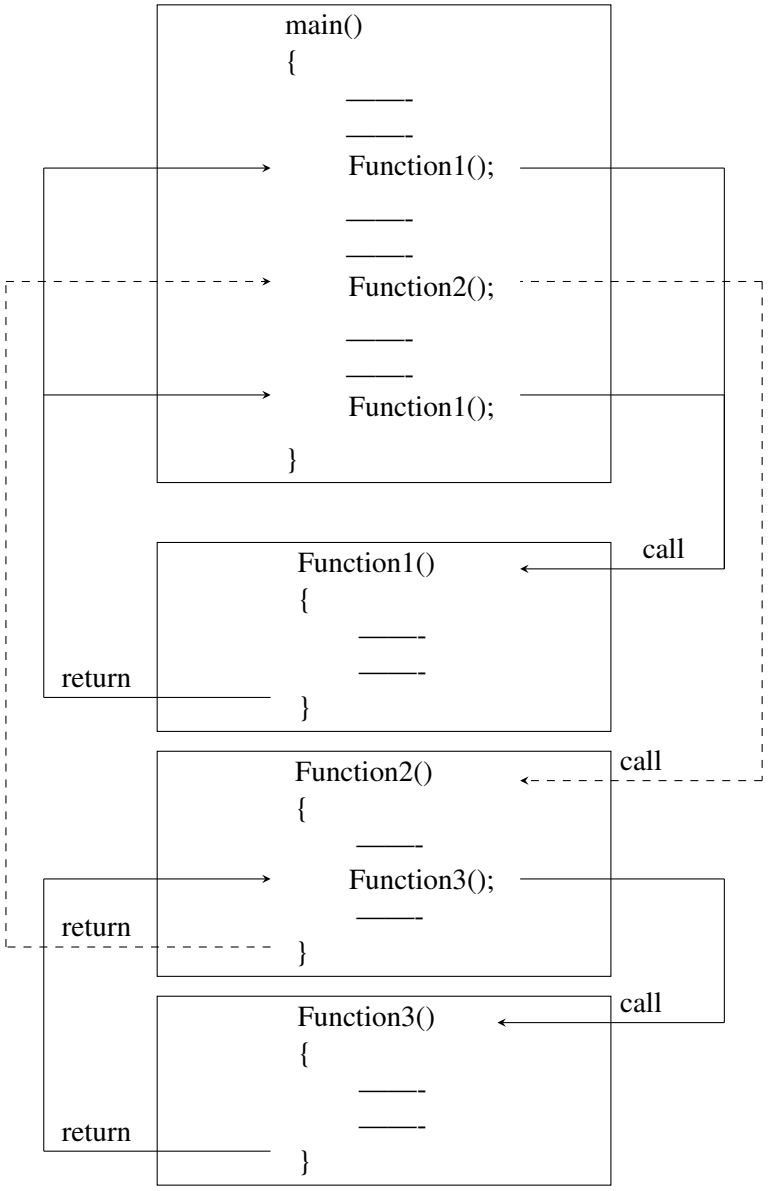

Figure 1.2: Program with multiple user defined functions

- Any function call any other function.
- A called function can also call another function.
- A function can be called more than once.

1.2 The Form of C Functions

To make user-defined functions, the following parts are to be established.

1. Function Definition
2. Function call
3. Function Declaration (Function Prototype)

1.2.1 Definition of Functions

General format of function definition is given below.

```
function_type function_name(paramater list)
{
  local variable declaration;
  executable statement1;
  executable statement2;
  ....
  ....
  return statement;
}
```

Listing 1.1: General format of a function definition

A **function definition** (**function implementation**) shall include the following elements:

1. Function header
2. Function body

Function Header

Function header consists of three parts:

- **Function type (Return type):**
  The function type specifies the type of value the function is expected to return to calling function. If return type is not specified, C will assume that it is an integer type. If the function does not return anything, then return type should be specified as `void`.
- **Function name:**
  Function name is any valid C identifier and therefore must follow the same rules for variable names.
- **Formal parameter list:**
  Formal parameter list declares the variable that will receive the data sent by the calling function. The parameter list contains the declaration of variables separated by commas and surrounded by parentheses.

  ```
  float volume(int r, int h)
  ```

  Listing 1.2: Formal parameter list

  If the function does not receive any values from the calling function, then an empty pair of parentheses has to be put.

  ```
  float welcome()
  ```

  Listing 1.3: Example for function without parameters

Function Body

A function body contains three parts

1. Local declarations that specify the variables needed by the function.
2. Function statements that perform the task of the function.
3. A `return` statement that returns the value evaluated by the function. The `return` statement can be omitted if the function does not return any value. The `return` statement can take the following form:

```
return(expression);
```

Listing 1.4: Syntax of `return` statement

When a `return` is encountered, the control is immediately passed back to the calling function. An example for a complete function definition is given below

```
float findArea(float l, float b)
{
  float area;
  area = l * b;
  return (area);
}
```

Listing 1.5: Example for function definition

1.2.2 Function Call

A function can be called by using the function name followed by a list of **actual parameters (or arguments)**, if any, enclosed in parentheses. For example:

```
void main()
{
  float l,b,area;
  area = findArea(10.2,20.5); /*Function Call*/
  printf("Area= %f",area);
}
```

Listing 1.6: Example for function call

When the function call occurs, the control is transferred to the function `findArea()`. The function is then executed and a value is returned when a `return` statement is encountered. This value is assigned to the variable `area`.

1.2.3 Function Declaration (Function Prototype)

Like variables, all functions in C program must be declared, before they are invoked. A function declaration consists of 4 parts.

1. Function Type
2. Function Name
3. Parameter List
4. Terminating semicolon

Note: Refer the Function Header section

General format of a function declaration is:

```
function_type function_name (paramter_lst);
```

Listing 1.7: General format of function declaration

An example for function declaration is given below:

```
float findArea(float l, float b);
```

Listing 1.8: Example for function declaration

A prototype declaration can be placed in two places in a program.

1. Above all the functions including the `main()` function.
   In this case, the prototype is referred to as **global prototype**. Such declarations are available for all the functions in the program.
2. Inside a function definition
   In this case, the prototype is called **local prototype**. Such declarations are only available for that function.

1.3 Category of Functions

Depending on whether a function takes up arguments and whether a function returns a value, functions can be classified into different categories

1.3.1 No Arguments and No Return Value

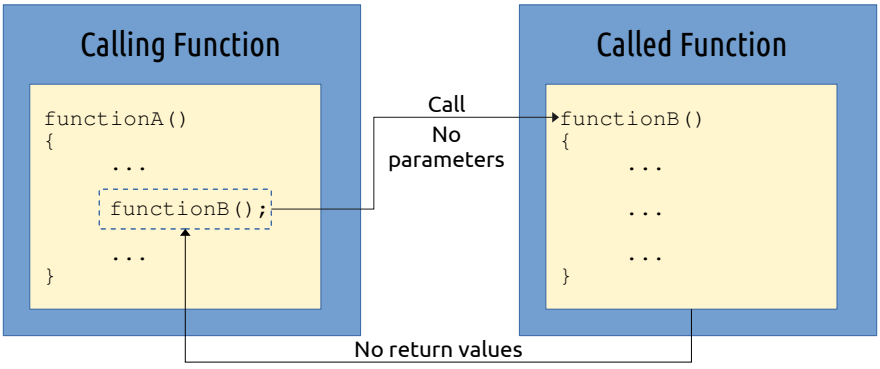

Figure 1.3: No arguments and no return values

In this case,

- The called function does not receive any value from the calling function.
- The called function does not return any value to the calling function.

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

int main()
{
  printf("Area and Perimeter of Rectangle\n");
  /*calling function without passing arguments*/
  findAreaPerimeter();
}

void findAreaPerimeter()
{
  float l,b,area;
  printf("Enter the length and breadth: ");
  scanf("%f%f",&l,&b);
  area=l*b;
  printf("Area = %f\n",area);
}
```

Listing 1.9: Example for function with no arguments and no return value

1.3.2 Arguments but No Return Value

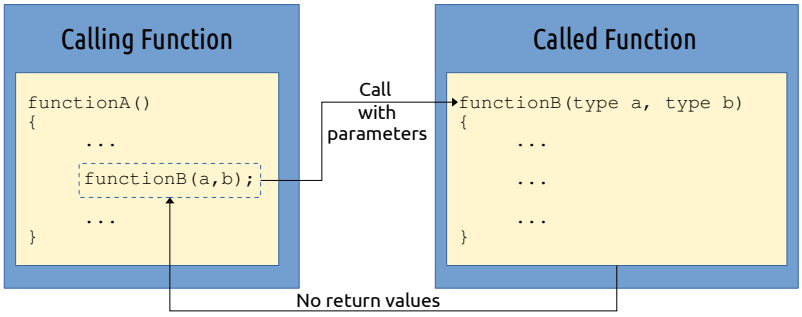

Figure 1.4: Arguments but no return value

In this case,

- The calling function passes one or more arguments to the called function.
- The called function does not return any value to the calling function.

```c
#include<stdio.h>
void findAreaPerimeter(float,float);
int main()
{
  float l,b;
  printf("Enter the length and breadth: ");
  scanf("%f%f",&l,&b);
  /*calling function by passing parameters*/
  findAreaPerimeter(l,b);
}
void findAreaPerimeter(float l, float b)
{
  float area;
  area=l*b;
  printf("Area = %f\n",area);
}
```

Listing 1.10: Example for function with arguments but no return value

1.3.3 Arguments with Return Value

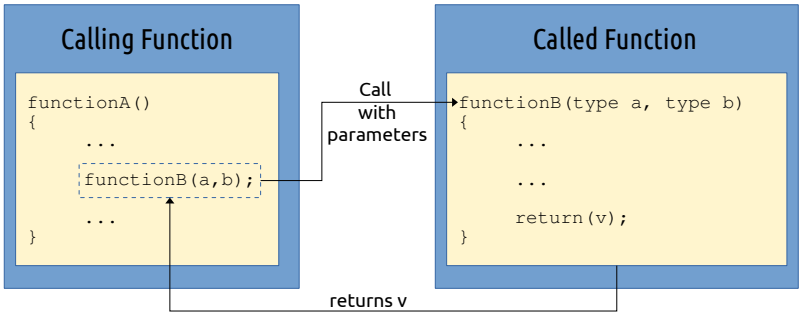

Figure 1.5: Arguments with return value

In this case,

- The calling function passes one or more arguments to the called function.
- The called function returns a value to the calling function.

```
#include<stdio.h>
float findAreaPerimeter(float,float);
int main()
{
  float l,b,area;
  printf("Enter the length and breadth: ");
  scanf("%f%f",&l,&b);
  /*calling function by passing parameters*/
  area = findAreaPerimeter(l,b);
  printf("Area=%f\n",area);
}
float findAreaPerimeter(float l, float b)
{
  float area;
  area=l*b;
  return(area);
}
```

Listing 1.11: Example for function with arguments and a return value

1.4 Nesting of Functions

In C, function calls can be nested. i.e. a function can call another function, and the called function can call another function and so on. An example is given below.

```
#include<stdio.h>
#define PI 3.14
float findCylinderVolume(float,float);
float findBaseArea(float);
int main()
{
  float r,h,volume;
  printf("Enter the radius and height of the cylinder: ");
  scanf("%f%f",&r,&h);
  /*calling function by passing parameters*/
  volume = findCylinderVolume(r,h);
  printf("Volume=%f\n",volume);
}

float findCylinderVolume(float r, float h)
{
  float volume,area;
  /*calling another function*/
  area=findBaseArea(r);
  volume=area*h;
  return(volume);
}

float findBaseArea(float r)
{
  float area;
  area=PI*r*r;
  return(area);
}
```

Listing 1.12: Example for nesting of functions

1.5 Recursion

Recursion is a process by which a function calls itself repeatedly until some specified condition has been satisfied. This process is used for repetitive computations in which each action is stated in terms of a previous result.

In order to solve a problem recursively, two conditions must be satisfied.

1. The problem must be written in recursive form
2. The problem statement must include a stopping condition

For example to find the factorial of a number, the problem can be written as

$$n! = n * (n-1)!$$

Also we know that

$$1! = 1$$

Therefore

$$n! = n * (n-1)!$$

$$(n-1)! = (n-1) * (n-2)!$$

$$(n-2)! = (n-2) * (n-3)!$$

..

$$2! = 2 * 1!$$

This method can implemented using recursion as follows

```
#include<stdio.h>
long int factorial(int n);
void main()
{
  int n;
  long int f;
  printf("Enter the number:");
  scanf("%d",&n);
  f = factorial(n);
  printf("%d! = %ld\n",n,f);
}

long int factorial(int n)
{
  if(n<=1)
  {
    return 1;
  }
  else
  {
    //recursive call to the function factorial
    return (n*factorial(n-1));
  }
}
```

Listing 1.13: Factorial of a number using recursion

1.6 Storage Classes in C

In C, all variables have a data type and also a **storage class**. Storage classes in C are:

1. Automatic
2. External
3. Static
4. Register

The **scope** of a variable determines, over what region of the program a variable is actually available for use.

The **lifetime (longevity)** of a variable refers to the period during which a variable retains a given value during execution of a program.

1.6.1 Automatic Variables

- Automatic variable are declared inside a function in which they are to be utilized.
- They are created when the function is called and destroyed automatically when the function is exited, hence the name **automatic**.
- Automatic variables are therefore local to the function in which they are declared. Hence, automatic variables are also referred to as **local** or **internal** variables.
- Life time of an automatic variable will be till the function lasts.
- Automatic variable will have some garbage value if not initialized explicitly.
- A variable declared inside a function without storage class specification is, by default, an automatic variable. For example:

```
int main()
{
  int n;   /* declares n as an automatic variable */
  auto int a; /* declares a as an automatic variable */
}
```

Listing 1.14: Declaration of automatic variables

1.6.2 External Variables

- Variables that are both alive and active thorughout the entire program are known as **external variables (global variables)**
- Unlike local variables, global variables can be accessed by any function in the program
- Global variable will be initialized to 0 if not initialized explicitly
- External variables are declared outside all functions. For example, external declaration of integer `number` and float `length` might appear as

```
int number;
float length = 7.5;
main()
{
  -----
}
function1()
{
  -----
}
function2()
{
  -----
}
```

Listing 1.15: External variables declaration

Here, the variables `number` and `length` are available for use in all the three functions.

- Changes made to an external variable by one function will be visible to all the functions in the program.
- In case a local variable and global variable has the same name, the local variable will have precedence over the global variable in the function where it is declared. An example given below illustrates this

```
#include<stdio.h>
//Declaration Global variable x
int x=10;
void sampleFun();
int main()
{
  x++;
  /*Prints the value of global variable x = 11*/
  printf("Value of x in main = %d\n",x);
  sampleFun();
}
void sampleFun()
{
  //Declaration local variable x
  int x=20;
  x++;
  /*Local variable x gets precedence over global variable x */
  /*Prints the value of x as 21*/
  printf("Value of x in sampleFun = %d\n",x);
}
```

Listing 1.16: Example for external and local variables with same name

External Declaration

If an external variable is declared after a function, then the variable must be declared inside the function using the storage class specifier **extern**. For example:

```
void fun1();
int main()
{
  extern int i; /* External Declaration */
  ------
  ------
}
void fun1()
{
  extern int i; /* External Declaration */
  ------
  ------
}
int i;
```

Listing 1.17: Declaration of external variable with `extern` keyword

1.6.3 Static Variables

- Static variables persists until the end of the program
- A variable can be declared static by using the keyword **static**
- A static variable may be either an internal or external type depending upon the place of declaration
- Internal static variable are declared inside a function. Internal static variables are similar to automatic variables, except that they remain in existence throughout the remainder of the

program. Therefore internal static variables can be used to retain values between function calls.

- A static variable is initialized only once, when the program is compiled
- An external static variable is declared outside all the functions and is available to all the functions in that program. The difference between static external variable and simple external variable is that the static external variable is available only within the file where it is defined while the simple external variable can be accessed by other file.
- Static variable will be initialized to 0 if not initialized explicitly.
- An example differentiating static and automatic variable is given below.

```
void display();
void main()
{
  dipslay();
  display();
}
void display()
{
  int i = 0;
  i++;
  printf("\n%d",i);
}

/*Prints output as
1
1
*/
```

Listing 1.18: Automatic variable `i`

```
void display();
void main()
{
  dipslay();
  display();
}
void display()
{
  static int i = 0;
  i++;
  printf("\n%d",i);
}

/*Prints output as
1
2
*/
```

Listing 1.19: Static variable `i`

1.6.4 Register Variables

- We can tell the compiler that a variable should be kept in one of the machine's registers, instead of keeping in the memory
- Since a register access is much faster than a memory access, keeping the frequently accessed variables in the register will lead to faster execution of the programs
- Register variable will have garbage initial value. Register variables can be declared as follows

```
register int i;
```

Listing 1.20: Declaration of a register variable

- Register variables are local to the function in which they are declared

1.6.5 Summarizing Storage Classes

Automatic

Keyword	: `auto`
Initial value	: Garbage
Storage location	: RAM
Scope	: Local to the block where the variable is defined
Lifetime	: As long as the control is within the block where the variable is declared

External

Keyword	: `extern`
Initial value	: 0
Storage location	: RAM
Scope	: Entire program
Lifetime	: As long as the program is under execution

Static

Keyword	: `static`
Initial value	: 0
Storage location	: RAM
Scope	: Local to the block where the variable is defined
Lifetime	: As long as the program is under execution

Register

Keyword	: `register`
Initial value	: Garbage
Storage location	: Register
Scope	: Local to the block where the variable is defined
Lifetime	: As long as the control is within the block where the variable is declared