FUNCTIONS

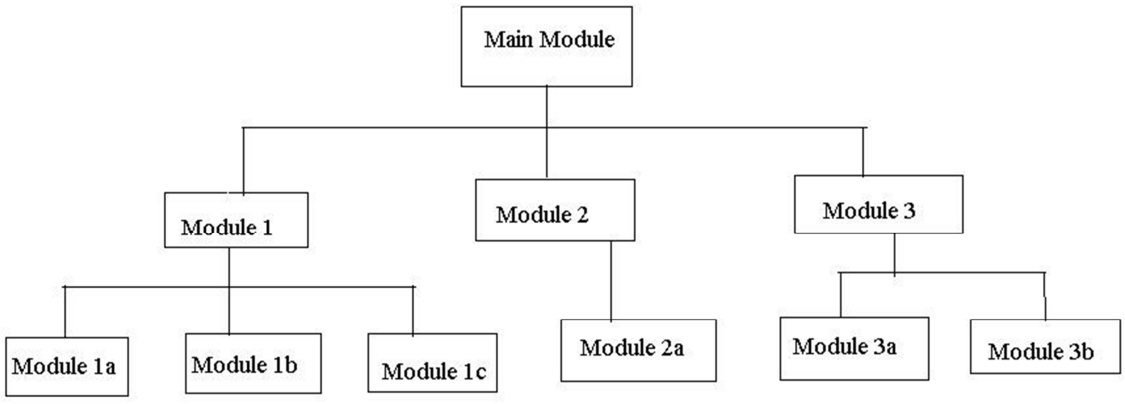
The planning for large programs consists of

- First understanding the problem as a whole,
- Second breaking it into simpler, understandable parts.

We call each of these parts of a program a **module** and the process of subdividing a problem into manageable parts is - **Top-Down Design**.

The principles of top-down design and structured programming dictate that a program should be divided into a main module and its related modules. Each module is in turn divided into sub-modules until the resulting modules are inseparable; i.e. until they are implicitly understood without further division.

Top-down design is usually done using a visual representation of the modules known as a **structure chart**. The structure chart shows the relation between each module and its sub-modules. The structure chart is read top-down, left-right. First we read Main Module. Main Module represents our entire set of code to solve the problem.



Structure Chart

Moving down and left, we then read Module 1. On the same level with Module 1 are Module 2 and Module 3. The Main Module consists of three sub-modules. At this level, however we are dealing only with Module 1. Module 1 is further subdivided into three modules, Module 1a, Module 1b, and Module 1c. To write the code for Module 1, we need to write code for its three sub-modules.

The Main Module is known as a **calling module** because it has sub-modules. Each of the sub-modules is known as a called module. But because Modules 1, 2, and 3 also have sub-modules, they are also calling modules; they are both called and calling modules.

Communication between modules in a structure chart is allowed only through a calling module. If Module 1 needs to send data to Module 2, the data must be passed through the calling module, Main Module. No communication can take place directly between modules that do not have a **calling-called relationship**.

How can Module 1a send data to Module 3b?

As depicted in the above **structure chart,** It first sends data to Module 1, which in turn sends it to the Main Module, which passes it to Module 3, and then on to Module 3b.

The technique used to pass data to a function is known as **parameter passing (or Message Passing)**. The parameters are contained in a list that is a definition of the data passed to the function by the caller.

- **A function is a self-contained program segment that carries out some specific well defined tasks. (Contains a set of code written to achieve a specific task).**

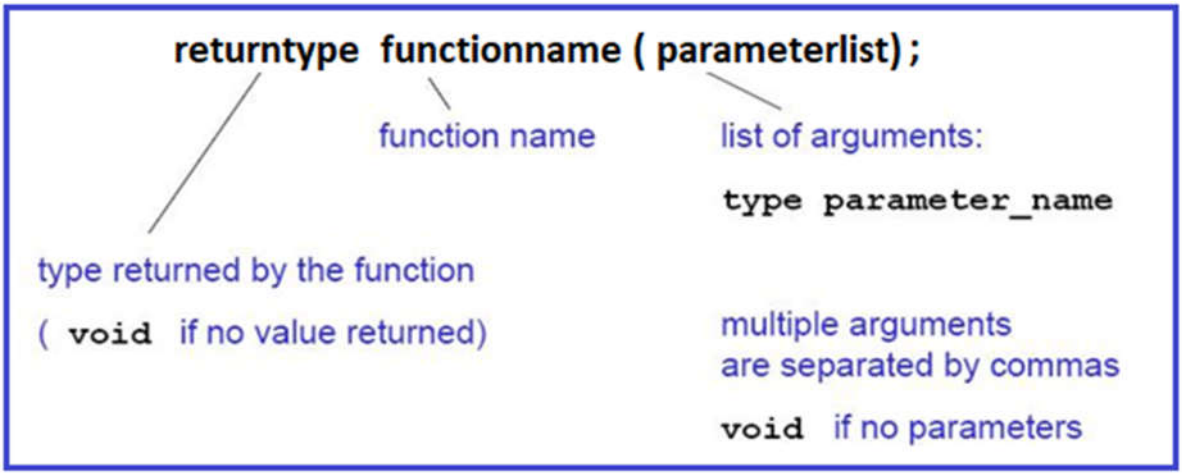
Advantages of functions

1. Code Reusability
2. Improves Program Readability
3. Divide a complex problem into simpler ones.
4. Modifying a program becomes easier
5. Minimizes the chances of error
6. Debugging a program is easier

In a C program, in order to write and use a function, a function must have:

1. Function Declaration (if required)
2. Function Definition
3. Function Calling

1. Function Declaration (Prototype)



Example:

```
int     add(int);
float   add(int, int, float);
```

```
void   add(int,int);
```

2. Function Definition

```
returntype     functionname( parameter list )
{
        declarations;

        statements;
}
```

Example:

```
1. int  add(int x)
    {
       x++;
       return x;
     }
```

Note: Use return keyword for returning value from a function.

```
2. void  add(int x)
    {
       printf(" Sum = %d", x++);
     }
```

3. Function Calling

```
functionname( parameter list );
```

Function Declaration (PROTOTYPES)

- **If** a function is not defined before it is used, it must be declared by specifying the return type and the types of the parameters.

  double sqrt(double);

  Tells the compiler that the function **sqrt()** takes an argument of type double and returns a double. This means, that variables will be cast to the correct type. So **sqrt(4)** will return the correct value even though **4** is **int** not **double**. These function prototypes are placed at the top of the program, or in a separate header file, **file.h**, included as

  #include "file.h"

  Variable names in the argument list of a function declaration are optional:

  void f (char, int);

  void f (char c, int i); /*equivalent but makes code more readable */

- If all functions are defined before they are used, no prototypes are needed. In this case, **main()** is the last function of the program.

Scope rules for Functions

Variables defined within a function (including main) are local to this function and no other function has direct access to them. The only way to pass variables to function is as parameters. The only way to pass (a single) variable back to the calling function is via the return statement.

Example:

```
#include<stdio.h>
int func (int n)
{
 printf("%d\n",b);
 return n;
}//b not defined locally!

int main (void)
{
 int a = 2, b = 1, c;
 c = func(a);
 return 0;
}
```

The above code gives error after compilation

```
error: 'b' undeclared (first use in this function)
```

Because variable b is defined only in main() & thus scope of variable b is limited within main() only.

Function Call

When a function is called, expressions in the parameter list are evaluated (in no particular order!) and results are transformed to the required type. Parameters are copied to local variables for the function and function body is executed when return is encountered, the function is terminated and the result (specified in the return statement) is passed to the calling function (for example main).

Example:

```
#include<stdio.h>
int fact (int n)
{
 int i, product = 1;
  for (i = 2; i <= n; ++i)
      product *= i;
 return product;
}
int main (void)
{
 int i = 12;
 printf("%d",fact(i));
 return 0;
}
```

Types of Functions

1. **User-defined Functions**
2. **Library Functions**

User-defined Functions

Every program must have a main function to indicate where the program has to begin its execution. While it is possible to code any program utilizing only main function, it leads to a number of problems. The program may become too large and complex and as a result task of debugging, testing and maintaining becomes difficult.

If a program is divided into functional parts, then each part may be independently coded and later combined into a single unit, these subprograms called “functions” are much easier to understand debug and test.

There are times when some types of operation for calculation are repeated many times at many points throughout a program.

For instance, we might use the factorial of a number at several points in the program. In such situations, we may repeat the program statements whenever they are needed. Another approach is to design a function that can be called and used whenever required.

Once a function has been designed and packed it can be treated as a “black box” that takes some data from main program and returns a value. The inner details of the program are invisible to the rest of program.

The return statement is the mechanism for returning a value to the calling function.

We can tell a function to return a particular type of data by using a type-specifier in the header (function declaration);

A function can be called by simply using the function name in the statement.

Standard Library Functions and Header files:

C functions can be classified into two categories,

1. Library functions
2. User-defined functions.

main() is the example of user-defined functions.
printf()and scanf() belong to the category of library functions.

The main difference between these two categories is that library functions are not required to be written by us where as a user-defined function has to be developed by the user (developer) at the time of writing a program. However, a user-defined function can later become a part of the c program library.

ANSI C Standard Library Functions:

The C language is accompanied by a number of library functions that perform various tasks. The ANSI committee has standardized header files which contain these functions.

Some of the Header files are:

<ctype.h>	**character testing and conversion functions.**
<math.h>	**Mathematical functions**
<stdio.h>	**standard I/O library functions**
<stdlib.h>	**Utility functions such as string conversion routines memory allocation, random number generator etc.**
<string.h>	**string Manipulation functions**
<time.h>	**Time Manipulation functions**

MATH.H

The math library contains functions for performing common mathematical operations.

Some of the functions are :

abs : returns the absolute value of an integer x

cos : returns the cosine of x, where x is in radians

exp: returns "e" raised to a given power

fabs: returns the absolute value of a float x

log: returns the logarithm to base e

log10: returns the logarithm to base 10

pow : returns a given number raised to another number

sin : returns the sine of x, where x is in radians

sqrt : returns the square root of x

tan : returns the tangent of x, where x is in radians

ceil : The ceiling function rounds a number with a decimal part up to the next highest integer (written mathematically as $\lceil x \rceil$)

floor : The floor function rounds a number with a decimal part down to the next lowest integer (written mathematically as $\lfloor x \rfloor$)

STRING.H

There are many functions for manipulating strings. Some of the more useful are:

strcat : Concatenates (i.e., adds) two strings

strcmp: Compares two strings

strcpy: Copies a string

strlen: Calculates the length of a string (not including the null)

strstr: Finds a string within another string

strtok: Divides a string into tokens (i.e., parts)

STDIO.H

printf: Formatted printing to stdout

scanf: Formatted input from stdin

fprintf: Formatted printing to a file

fscanf: Formatted input from a file

getc: Get a character from a stream (e.g, stdin or a file)

putc: Write a character to a stream (e.g, stdout or a file)

fgets: Get a string from a stream

fputs: Write a string to a stream

fopen: Open a file

fclose: Close a file

STDLIB.H

atof: Convert a string to a double (not a float)

atoi: Convert a string to an int

exit: Terminate a program, return an integer value

free: Release allocated memory

malloc: Allocate memory

rand: Generate a pseudo-random number

system: Execute an external command

TIME.H

This library contains several functions for getting the current date and time.

time: Get the system time

ctime: Convert the result from time() to something meaningful

Categories of Functions

A function depending on whether arguments are present or not and whether a value is returned or not can be categorized as:

1. Without Argument and Without return value
2. Without Argument and With return value
3. With Arguments and Without return value.
4. With Arguments and With return value.

1. Without Argument and Without return value

When a function has no arguments, it does not receive any data from calling function. In effect, there is no data transfer between calling function and called function.

```c
#include<stdio.h>
void demo(void); // function declaration
void main()
{
   demo();//function calling
}
void demo()//function definition
{
   printf(" User-defined category of Function: Without Argument and Without return value");
}
```

2. Without Argument and With return value

The function doesn't accept argument but return a value.

```c
#include<stdio.h>
int demo(void); // function declaration-Without Argument and With return value
void main()
{
   int result;
   result = demo();//function calling
   printf(" From demo(), Value received:  %d", result);
}
int demo()//function definition
{
   return(12);
}
```

Note: Data type of returning value must be same (typecasted) as specified in the function declaration.

In the above program 12 is considered as an integer in return statement, because return type of demo() is declared as int.

3. With Arguments and Without return value.

The function takes argument but does not return value.

The actual argument (sent through main) and formal argument (declared in header section) should match in number, type and order.

In case, actual arguments are more than formal arguments, extra actual arguments are discarded. On the other hand unmatched formal arguments are initialized to some garbage values.

```
1  #include<stdio.h>
2  void demo(int, int); // function declaration-With Argument and Without return value
3  void main()
4  {
5     demo(12);//function calling
6  }
7  void demo(int x, int y)//function definition
8  {
9     printf(" From function, Addition of values received: %d", (x+y));
10 }
11
```

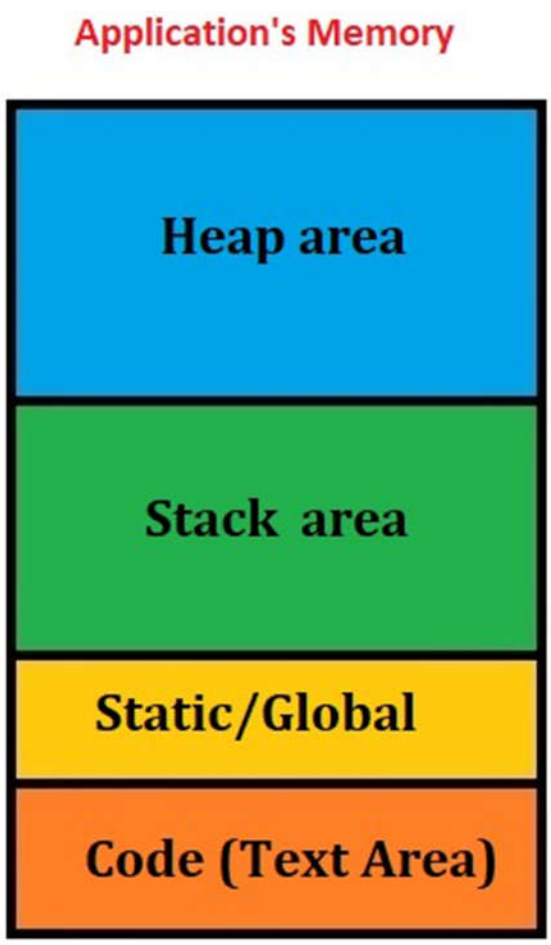
4. With Arguments and With return value.

The function accepts arguments and return value.

```
1  #include<stdio.h>
2  float demo(float, float); // function declaration-With Arguments and With return value
3  void main()
4  {
5     float result;
6     result = demo(12.78 , 67.8);//function calling
7     printf(" From function, Addition of values received: %f", result);
8  }
9  float demo(float x, float y)//function definition
10 {
11    return (x+y);
12 }
13
```

How C Program is executed in Memory?

Every application gets some space in memory (RAM) for its execution.



The memory allocated to an application/ program is typically categorized in the four segments:

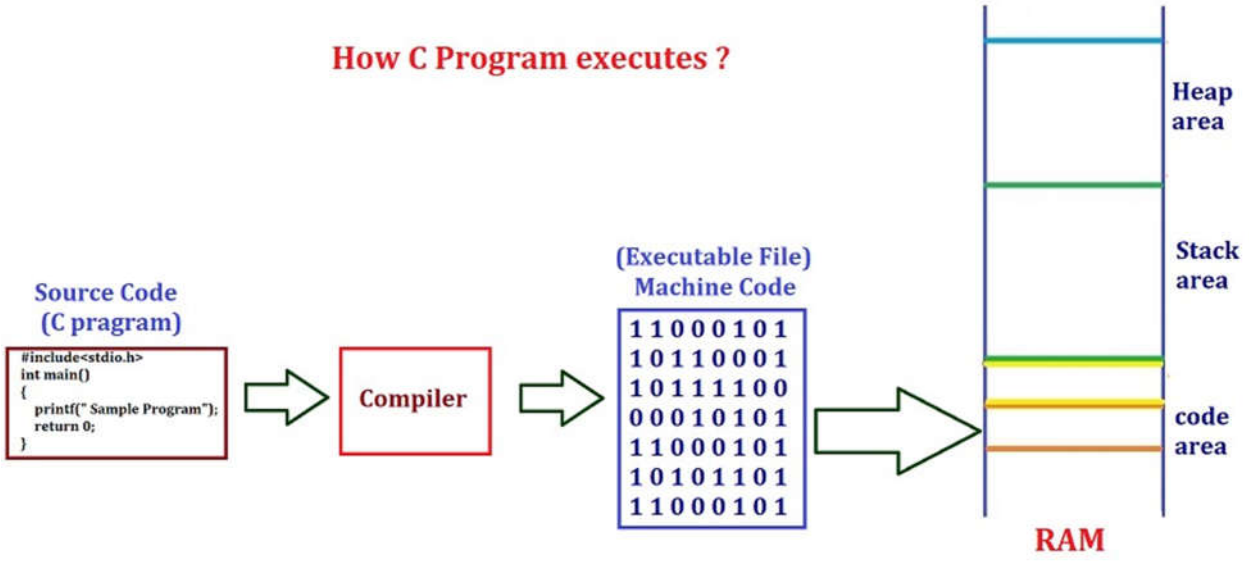
1. **Heap Area**
2. **Stack Area**
3. **Static/ Global**
4. **Code(Text Area)**

Heap Area: Memory is allocated at the time of execution from this area (called as dynamic memory allocation).

Stack Area: Here memory is allocated for the function calls. For each function call, a stack frame (also called as Activation Record) is created. Soon after completion of the function related stack frame (activation record) is deleted.

Static/Global: Memory is allocated for any static or global variable from this area.

Code (Text Area): Contains the equivalent machine code of the program to be executed.



At lowest level in computer's architecture, it understands and executes only **binary instructions**. Any instruction that has to be executed by the computer has to be encoded in binary (based on encoding rules).

A program written in C language (high-level language) passed to a program (software) called **Compiler** that generates corresponding machine-level/ executable code which is instructions encoded in binary.

Parameter Passing Techniques

1. Call by Value
2. Call by Reference.

1. Call by Value

The process of passing the actual value of variables is known as **Call by Value**.

Example:

```
1   #include<stdio.h>
2   #include<conio.h>
3   void add(int);//function declaration
4   void main()
5   {
6      int a=1432;
7      add(a);//function calling
8      printf("\n a= %d", a);
9      getch();
10  }
11  void add(int x)//function definition
12  {
13    x = x +1;
14  }
```

2. Call by Reference.

The process of calling a function using pointers to pass the addresses of variables is known as **Call by Reference.**

The function which is called by reference can change the value of the variable used in the call.

Example:

```c
#include <stdio.h>
void swap(int *,int *);//function declaration
int main()
{
    int a,b;
    printf("Enter the Values of a and b:");
    scanf("%d%d",&a,&b);
    printf("\n Before Swapping \n");
    printf("a = %d \t b = %d", a,b);
    swap(&a,&b); //function calling
    printf("\nAfter Swapping \n");
    printf("a = %d \t b = %d", a,b);
    return 0;
}
void swap(int *x, int *y)//function definition
{
    int temp;
    temp = *x;
    *x = *y;
    *y = temp;
}
```