FUNCTIONS

- A function is a group of statements that together perform a specific task when it is called.
- Every C program has at least one function, which is **main()**
- The function contains the set of programming statements enclosed by { }.

Advantags

- By using functions, we can avoid rewriting same logic/code again and again in a program.
- We can call functions any number of times in a program and from any place in a program.
- We can track a large C program easily when it is divided into multiple functions.
- Reusability is the main achievement of C functions.
- However, Function calling is always a overhead in a C program.

Function Aspects

There are three aspects of a C function.

- **Function declaration** A function must be declared globally in a c program to tell the compiler about the function name, function parameters, and return type.
- **Function call** Function can be called from anywhere in the program. The parameter list must not differ in function calling and function declaration. We must pass the same number of functions as it is declared in the function declaration.
- **Function definition** It contains the actual statements which are to be executed. It is the most important aspect to which the control comes when the function is called. Here, we must notice that only one value can be returned from the function.

SN	C function aspects	Syntax

1	Function declaration	return_type function_name (argument list);
2	Function call	function_name (argument_list)
3	Function definition	return_type function_name (argument list) {function body;}

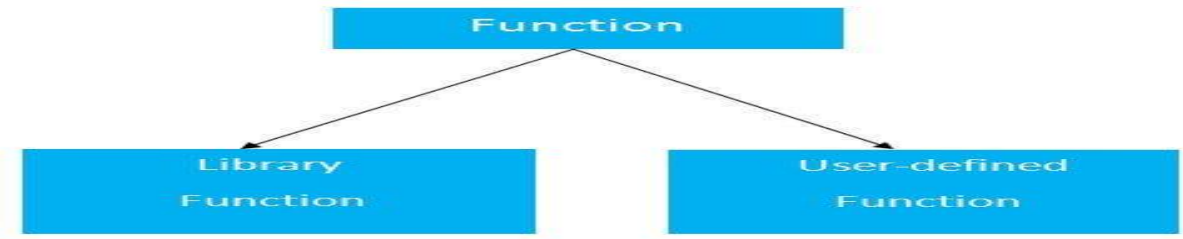
Syntax :

```
return_type function_name(data_type parameter...)
{
//code to be executed
}
```

Types of Functions

There are two types of functions in C programming:

1. **Library Functions/Pre defined/Built-in Functions**: are the functions which are declared in the header files such as scanf(), printf(), gets(), puts(), ceil(), floor() etc.
2. **User-defined functions**: are the functions which are created by the programmer, so that he/she can use it many times. It reduces the complexity of a big program and optimizes the code.



Return Value

A function may or may not return a value from the function. If you don't have to return any value from the function, use void for the return type.

Let's see a simple example of C function that doesn't return any value from the function.

Example without return value:

```
void hello()
   {
printf("hello c");
}
```

- If you want to return any value from the function, you need to use any data type such as int, long, char, etc.
- The return type depends on the value to be returned from the function.

Let's see a simple example of C function that returns int value from the function.

Example with return value:

```
int get()
{
return 10;
}
```

In the above example, we have to return 10 as a value, so the return type is int. If you want to return floating-point value (e.g., 10.2, 3.1, 54.5, etc), you need to use float as the return type of the method.

```
float get()
{
return 10.2;
}
```

Now, you need to call the function, to get the value of the function.

Different aspects of function calling

- A function may or may not accept any argument.
- It may or may not return any value.
- Based on these facts,

There are four different aspects of function calls.

1. function without arguments and without return value
2. function without arguments and with return value
3. function with arguments and without return value

4. function with arguments and with return value

Example: Function without argument and without return value

Example :

```
#include<stdio.h>
void printName();  //Function declaration
void main ()
{
   printf("Hello ");
   printName();  //Function call
}
void printName()  //Function Defination
{
   printf("SIDDARTHA");
}
```

Output: Hello SIDDARTHA

Example :

```
#include<stdio.h>
void sum();
void main()
   {
 printf("\ Calculate the sum of two numbers:");
 sum();
}
void sum()
{
   int a,b;
   printf("\nEnter two numbers");
   scanf("%d %d",&a,&b);
   printf("The sum is %d",a+b);
}
```

Output:

Calculate the sum of two numbers:

Enter two numbers10

20

The sum is 30

Example for Function without argument and with return value

Example:

```
#include<stdio.h>
int sum();
void main()
{
   int result;
   printf("\n Calculate the sum of two numbers:");
   result = sum();
   printf("%d",result);
}
int sum()
{
   int a,b;
   printf("\nEnter two numbers");
   scanf("%d %d",&a,&b);
   return a+b;
}
```

Output:

```
Calculate the sum of two numbers:
Enter two numbers30
60
90
```

Example : program to calculate the area of the square

```
#include<stdio.h>
int square();
void main()
{
   printf("Calculate the area of the square\n");
```

```
    float area = square();
    printf("The area of the square: %f\n",area);
}
int square()
{
    float side;
    printf("Enter the length of the side in meters: ");
    scanf("%f",&side);
    return side * side;
}
```

Output:

```
calculate the area of the square
Enter the length of the side in meters: 9
The area of the square: 81.000000
```

Example for Function with argument and without return value

Example:

```
#include<stdio.h>
void sum(int, int);
void main()
{
    int a,b,result;
    printf("\n Calculate the sum of two numbers:");
    printf("\nEnter two numbers:");
    scanf("%d %d",&a,&b);
    sum(a,b);
}
void sum(int a, int b)
{
    printf("\nThe sum is %d",a+b);
}
```

Output:

```
Calculate the sum of two numbers:
```

```
Enter two numbers:90
90
The sum is 180
```

Example : program to calculate the average of five numbers.

```
#include<stdio.h>
void average(int, int, int, int, int);
void main()
{
   int a,b,c,d,e;
   printf("\n Calculate the average of five numbers:");
   printf("\nEnter five numbers:");
   scanf("%d %d %d %d %d",&a,&b,&c,&d,&e);
   average(a,b,c,d,e);
}
void average(int a, int b, int c, int d, int e)
{
   float avg;
   avg = (a+b+c+d+e)/5;
   printf("The average of given five numbers : %f",avg);
}
```

Output:

```
Calculate the average of five numbers:
Enter five numbers:10
20
30
40
50
The average of given five numbers : 30.000000
```

Example for Function with argument and with return value

Example :

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

```
int sum(int, int);
void main()
{
   int a,b,result;
   printf("\n Calculate the sum of two numbers:");
   printf("\nEnter two numbers:");
   scanf("%d %d",&a,&b);
   result = sum(a,b);
   printf("\nThe sum is : %d",result);
}
int sum(int a, int b)
{
   return a+b;
}
```

Output:

```
Calculate the sum of two numbers:
Enter two numbers:30
60
The sum is : 90
```

Example 2: Program to check whether a number is even or odd

```
#include<stdio.h>
int even_odd(int);
void main()
{
 int n,flag=0;
 printf("\n To check whether a number is even or odd");
 printf("\nEnter the number: ");
 scanf("%d",&n);
 flag = even_odd(n);
 if(flag == 0)
 {
   printf("\nThe number is odd");
 }
 else
```

```
 {
   printf("\nThe number is even");
 }
}
int even_odd(int n)
{
   if(n%2 == 0)
   {
      return 1;
   }
   else
   {
      return 0;
   }
}
```

Output: To check whether a number is even or odd

Enter the number: 50

The number is even

SCOPE OF A VARIABLE

A scope in any programming is a region of the program where a defined variable can have its existence and beyond that variable it cannot be accessed. There are three places where variables can be declared in C programming language –

- Inside a function or a block which is called **local** variables.
- Outside of all functions which is called **global** variables.
- In the definition of function parameters which are called **formal** parameters.

Let us understand what are **local** and **global** variables, and **formal** parameters.

Local Variables

Variables that are declared inside a function or block are called local variables. They can be used only by statements that are inside that function or block of code. Here all the variables a, b, and c are local to main() function.

```
#include <stdio.h>
int main ()
  {
 /* local variable declaration */
 int a, b;
 int c;
  /* actual initialization */
 a = 10;
 b = 20;
 c = a + b;
  printf ("value of a = %d, b = %d and c = %d\n", a, b, c);
  return 0;
}
```

Output: value of a = 10, b = 20 and c = 30

Global Variables

Global variables are defined outside a function, usually on top of the program. Global variables hold their values throughout the lifetime of your program and they can be accessed inside any of the functions defined for the program.

A global variable can be accessed by any function. That is, a global variable is available for use throughout your entire program after its declaration. The following program show how global variables are used in a program.

```
#include <stdio.h>
/* global variable declaration */
int g;
int main ()
{
 /* local variable declaration */
 int a, b;
 /* actual initialization */
 a = 10;
 b = 20;
```

```
 g = a + b;
 printf ("value of a = %d, b = %d and g = %d\n", a, b, g);
 return 0;
}
```
Output: value of a = 10, b = 20 and g = 30

Formal Parameters/arguments

Formal parameters, are treated as local variables with-in a function and they take precedence over global variables.

Following is an example –

```
#include <stdio.h>
 /* global variable declaration */
int a = 20;
int main () {
 /* local variable declaration in main function */
 int a = 10;
 int b = 20;
 int c = 0;
 printf ("value of a in main() = %d\n",  a);
 c = sum( a, b);
 printf ("value of c in main() = %d\n",  c);
 return 0;
}
/* function to add two integers */
int sum(int a, int b) {
  printf ("value of a in sum() = %d\n",  a);
  printf ("value of b in sum() = %d\n",  b);
  return a + b;
}
```
Output:
value of a in main() = 10
value of a in sum() = 10
value of b in sum() = 20
value of c in main() = 30

RECURSION IN C

Recursion is the process of repeating items in a self-similar way. In programming languages, if a program allows you to call a function inside the same function, then it is called a recursive call of the function.

```
void recursion()
{
  recursion(); /* function calls itself */
}
int main() {
  recursion();
}
```

The C programming language supports recursion, i.e., a function to call itself. But while using recursion, programmers need to be careful to define an exit condition from the function, otherwise it will go into an infinite loop.

```
#include <stdio.h>
int fibonacci(int i)
  {
  if(i == 0)
    {
    return 0;
  }
  if(i == 1)
   {
    return 1;
  }
  return fibonacci(i-1) + fibonacci(i-2);
}
int  main()
{
  int i;
  for (i = 0; i < 10; i++)
   {
    printf("%d\t\n", fibonacci(i));
  }
  return 0;
}
```

Output:

```
0
1
1
```

```
2
3
5
8
13
21
34
```

Structures

- Structure is a user-defined data type
- A structure is a collection of one are more variables of different data types groped under a single name.
- Each element of a structure is called a member.
- The **struct** keyword is used to define the structure.

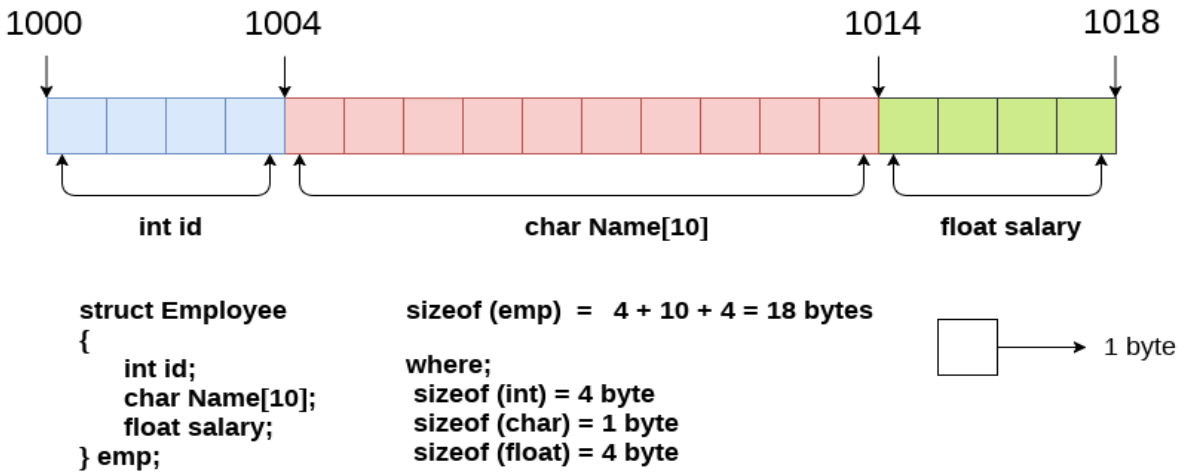
Let's see the syntax to define the structure .

```
struct structure_name
{
   data_type member1;
   data_type member2;
   ……..……
   data_type memeberN;
};
```

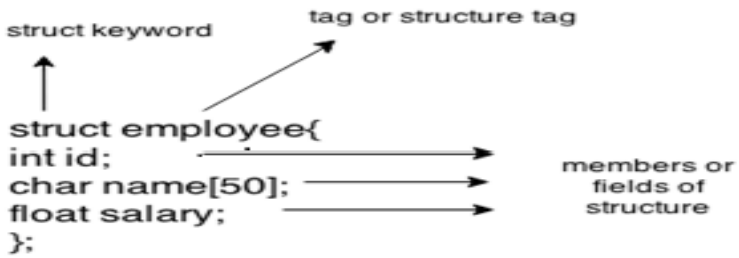
Let's see the example to define a structure for an entity employee.

```
struct employee
{
  int id;
  char name[10];
  float salary;
};
```

The following image shows the memory allocation of the structure employee that is defined in the above example.



Here, struct is the keyword; employee is the name of the structure; id, name, and salary are the members or fields of the structure. Let's understand it by the diagram given below:



Declaring structure variable

We can declare a variable for the structure so that we can access the member of the structure easily. There are two ways to declare structure variable:

1. By struct keyword within main() function
2. By declaring a variable at the time of defining the structure.

1st way:

Let's see the example to declare the structure variable by struct keyword. It should be declared within the main function.

```
struct employee
{
 int id;
  char name[50];
  float salary;
```

```
};  struct employee e1, e2;
```

Now write given code inside the main() function.

The variables e1 and e2 can be used to access the values stored in the structure.

2nd way:

Let's see another way to declare variable at the time of defining the structure.

```
struct employee
{   int id;
    char name[50];
    float salary;
}e1,e2;
```

Which approach is good

If number of variables are not fixed, use the 1st approach. It provides you the flexibility to declare the structure variable many times.

If no. of variables are fixed, use 2nd approach. It saves your code to declare a variable in main() function.

Accessing members of the structure

There are two ways to access structure members:

1. By . (member or dot operator)
2. By -> (structure pointer operator)

Example:

```
#include<stdio.h>
#include <string.h>
struct employee
{
    int id;
   char name[50];
}e1;  //declaring e1 variable for structure
```

```
int main( )
{
  //store first employee information
  e1.id=101;
  strcpy(e1.name, "Siddartha");//copying string into char array
  //printing first employee information
  printf( "employee 1 id : %d\n", e1.id);
  printf( "employee 1 name : %s\n", e1.name);
return 0;
}
```

Output:

```
employee 1 id : 101
employee 1 name : Siddartha
```

Let's see another example of the structure to store many employees information.

```
#include<stdio.h>
#include <string.h>
struct employee
{  int id;
   char name[50];
   float salary;
}e1,e2;  //declaring e1 and e2 variables for structure
int main( )
{
  //store first employee information
  e1.id=101;
  strcpy(e1.name, "abcde");//copying string into char array
  e1.salary=56000;

 //store second employee information
  e2.id=102;
  strcpy(e2.name, "xyz");
  e2.salary=126000;
```

```
  //printing first employee information
  printf( "employee 1 id : %d\n", e1.id);
  printf( "employee 1 name : %s\n", e1.name);
  printf( "employee 1 salary : %f\n", e1.salary);

  //printing second employee information
  printf( "employee 2 id : %d\n", e2.id);
  printf( "employee 2 name : %s\n", e2.name);
  printf( "employee 2 salary : %f\n", e2.salary);
  return 0;
}
```

Output:

```
employee 1 id : 101
employee 1 name : abcde
employee 1 salary : 56000.000000
employee 2 id : 102
employee 2 name : xyz
employee 2 salary : 126000.000000
```

Example:

```
#include <stdio.h>
#include <string.h>
struct Books
{
  char  title[50];
  char  author[50];
  char  subject[100];
  int   book_id;
};
int main( )
{
  struct Books Book1;       /* Declare Book1 of type Book */
  struct Books Book2;       /* Declare Book2 of type Book */
   /* book 1 specification */
```

```
   strcpy( Book1.title, "C Programming");
   strcpy( Book1.author, "ant");
   strcpy( Book1.subject, "C Programming Tutorial");
   Book1.book_id = 6495407;
   /* book 2 specification */
   strcpy( Book2.title, "Telecom Billing");
   strcpy( Book2.author, "abc");
   strcpy( Book2.subject, "Telecom Billing Tutorial");
   Book2.book_id = 6495700;
    /* print Book1 info */
   printf( "Book 1 title : %s\n", Book1.title);
   printf( "Book 1 author : %s\n", Book1.author);
   printf( "Book 1 subject : %s\n", Book1.subject);
   printf( "Book 1 book_id : %d\n", Book1.book_id);
   /* print Book2 info */
   printf( "Book 2 title : %s\n", Book2.title);
   printf( "Book 2 author : %s\n", Book2.author);
   printf( "Book 2 subject : %s\n", Book2.subject);
   printf( "Book 2 book_id : %d\n", Book2.book_id);
   return 0;
}
```

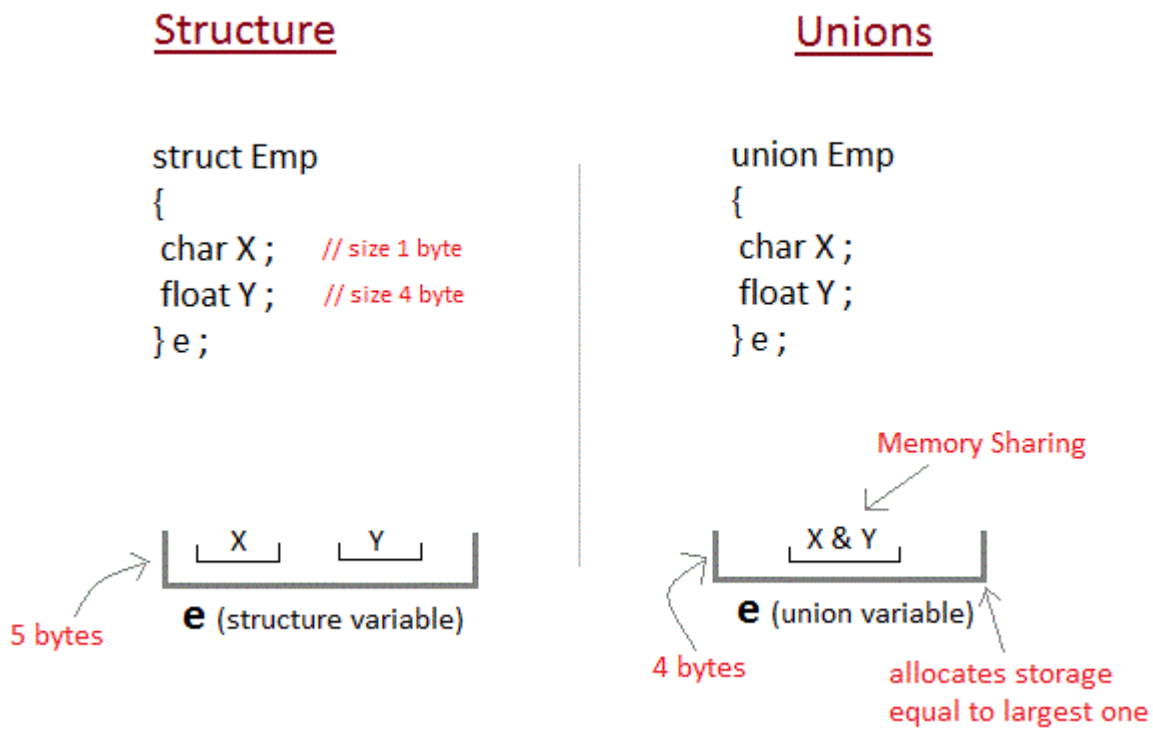
Output:

```
Book 1 title : C Programming
Book 1 author : ant
Book 1 subject : C Programming Tutorial
Book 1 book_id : 6495407
Book 2 title : Telecom Billing
Book 2 author : abc
Book 2 subject : Telecom Billing Tutorial
Book 2 book_id : 6495700
```

Unions

- Unions are conceptually similar to structures.
- A Union is a collection of one are more variables of different data types groped under a single name.
- The syntax to declare/define a union is also similar to that of a structure.

- The only differences is in terms of storage.
- In structure each member has its own storage location, whereas all members of union uses a single shared memory location which is equal to the size of its largest data member.



Example:

```
#include <stdio.h>
union unionJob
{
  //defining a union
  char name[32];
  float salary;
  int workerNo;
} uJob;

struct structJob
{
  char name[32];
  float salary;
  int workerNo;
```

```
} sJob;
int main()
{
  printf("size of union = %d bytes", sizeof(uJob));
  printf("\nsize of structure = %d bytes", sizeof(sJob));
  return 0;
}
```

Output:

size of union = 32 bytes

size of structure = 40 bytes

Pointers

- A pointer is a variable which stores the address of another variable.
- This variable can be of any type i.e. int, char, float…etc
- The size of the pointer depends on the architecture.
- However, in 32-bit architecture the size of a pointer is 2 byte.

How to Use Pointers

There are a few important operations, which we will do with the help of pointers very frequently.

(a) We define a pointer variable,

(b) assign the address of a variable to a pointer and

(c) finally access the value at the address available in the pointer variable.

This is done by using unary operator * that returns the value of the variable located at the address specified by its operand.

If we declare a variable v of type int, v will actually store a value.

int v=0;

v is equal to zero now.

However, each variable, apart from value, also has its address (or, simply put, where it is located in the memory).

The address can be retrieved by putting an ampersand (&) before the variable name.

&V

If you print the address of a variable on the screen, it will look like a totally random number (moreover, it can be different from run to run).

Example:

```
#include <stdio.h>
int main()
    {
   int v=0;
   printf("\n value of v variable v=%d",v);
   printf("\n Address of v variable v=%d",&v);
   return 0;
}
```

Output:

```
value of v variable v=0
Address of v variable v=1032003860
```

Declaring a pointer

The pointer in c language can be declared using * (asterisk symbol).

It is also known as indirection pointer.

```
int *a;//pointer to int
```

Example:

```
#include <stdio.h>
int main ()
   {
  int  v = 20;   /* actual variable declaration */
  int  *a;       /* pointer variable declaration */
  a = &v;  /* store address of v in pointer variable*/
  printf("\n value  of v variable: %d\n", v  );
  printf("Address of v variable: %d\n", &v  );
  /* address stored in pointer variable */
  printf("\n value of a variable: %d\n", a );
  /* access the value using the pointer */
     printf("Address of a variable: %d\n", &a  );
  printf("Value of *a variable: %d\n", *a );
```

```
  return 0;
}
```

Output:

```
value  of v variable: 20
Address of v variable: 1569213420
value of a variable: 1569213420
Address of a variable: 1569213424
Value of *a variable: 20
```

Types of Pointers in C

Following are the different Types of Pointers in C:

Null Pointer

- We can create a null pointer by assigning null value during the pointer declaration.
- This method is useful when you do not have any address assigned to the pointer.
- A null pointer always contains value 0.

Example: Null pointer:

```
#include <stdio.h>
int main()
  {
      int *p = NULL;        //null pointer
      printf("The value inside variable p is: %d",p);
      return 0;
}
```

Output:

```
The value inside variable p is: 0
```

Void Pointer

- A void pointer is also called as a generic pointer.
- It does not have any standard data type.
- A void pointer is created by using the keyword void.
- It can be used to store an address of any variable.

Example: Void Pointer

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

```
int main()
{
void *p = NULL;      //void pointer
printf("The size of pointer is:%d\n",sizeof(p));
return 0;
}
```

Output:

The size of pointer is:8

Pointer Arithmetic

- We can perform arithmetic operations on the pointers like addition, subtraction, etc.
- However, as we know that pointer contains the address, the result of an arithmetic operation performed on the pointer will also be a pointer if the other operand is of type integer.
- In pointer-from-pointer subtraction, the result will be an integer value.

Following arithmetic operations are possible on the pointers

- Increment
- Decrement
- Addition
- Subtraction

Incrementing Pointer

- If we increment a pointer by 1, the pointer will start pointing to the immediate next location.
- This is somewhat different from the general arithmetic since the value of the pointer will get increased by the size of the data type to which the pointer is pointing.
- We can traverse an array by using the increment operation on a pointer which will keep pointing to every element of the array, perform some operation on that, and update itself in a loop.

The Rule to increment the pointer is given below:

```
new_address= current_address + i * size_of(data type)
```

Where i is the number by which the pointer get increased.

32-bit : For 32-bit int variable, it will be incremented by 2 bytes.
64-bit : For 64-bit int variable, it will be incremented by 4 bytes.
Example: Incrementing pointer

```
#include<stdio.h>
int main()
{
int number=50;
int *p;//pointer to int
p=&number;//stores the address of number variable
printf("value of p variable is %u \n",p);
p=++p;
printf("After increment: Address of p variable is %u \n",p);
}
```

Output:

```
Address of p variable is 1228134268
After increment: Address of p variable is 1228134272
```

Example: Addition

```
#include<stdio.h>
int main()
{
int number=50;
int *p;//pointer to int
p=&number;//stores the address of number variable
printf("Address of p variable is %u \n",p);
p=p+1;
printf("After Addition: Address of p variable is %u \n",p); // in our case, p will get
 incremented by 4 byte return 0;
}
```

Output:

```
Address of p variable is 1326840556
After Addition: Address of p variable is 1326840560
```

Decrementing

- Like increment, we can decrement a pointer variable.
- If we decrement a pointer, it will start pointing to the previous location.

The formula of decrementing the pointer is given below:

new_address= current_address - i * size_of(data type)

Example: Decrementing pointer.

```
#include <stdio.h>
void main()
{
int number=50;
int *p;//pointer to int
p=&number;//stores the address of number variable
printf("Address of p variable is %u \n",p);
p=--p;
printf("After decrement: Address of p variable is %u \n",p); // P will now point to the immidiate previous location.
}
```

Output:

Address of p variable is 1783011372
After decrement: Address of p variable is 1783011368

Example: Subtraction

```
#include <stdio.h>
void main()
{
int number=50;
int *p;//pointer to int
p=&number;//stores the address of number variable
printf("Address of p variable is %u \n",p);
p=p-1;
printf("After Substraction: Address of p variable is %u \n",p); // P will now point to the immidiate previous location.
}
```

Output

Address of p variable is 2924723068
After Substraction: Address of p variable is 2924723064

Pointer to Pointer

- A pointer to a pointer is a form of multiple indirection, or a chain of pointers.
- Normally, a pointer contains the address of a variable.
- When we define a pointer to a pointer, the first pointer contains the address of the second pointer, which points to the location that contains the actual value as shown below.

- A variable that is a pointer to a pointer must be declared as such.
- This is done by placing an additional asterisk in front of its name.

For example, the following declaration declares a pointer to a pointer of type int

```
int **var;
```

Example:

```
#include <stdio.h>
 int main ()
{
   int  a;
   int  *b;
   int  **c;
   a = 3000;
   /* take the address of var */
   b = &a;
   /* take the address of ptr using address of operator & */
   c = &b;
   /* take the value using pptr */
   printf("\n Value of a = %d\n", a );
   printf("\n Value available at *b = %d\n", *b );
   printf("\n Value available at **c = %d\n", **c);
   return 0;
}
```

Output:

```
Value of a = 3000
Value available at *b = 3000
Value available at **c = 3000
```

Array of pointers

Before we understand the concept of arrays of pointers, let us consider the following example, which uses an array of 3 integers

Example:

```
#include <stdio.h>
const int MAX = 3;
int main ()
 {
  int  var[] = {10, 100, 200};
  int i;
    for (i = 0; i < MAX; i++
      {
    printf("Value of var[%d] = %d\n", i, var[i] );
  }
    return 0;
}
```

Output:

```
Value of var[0] = 10
Value of var[1] = 100
Value of var[2] = 200
```

There may be a situation when we want to maintain an array, which can store pointers to an int or any other data type .

Following is the declaration of an array of pointers to an integer −

```
int *ptr[MAX];
```

It declares **ptr** as an array of MAX integer pointers.

Thus, each element in ptr, holds a pointer to an int value.

Example:

```
#include <stdio.h>
const int MAX = 3;
int main ()
   {
  int  var[] = {10, 100, 200};
  int i, *ptr[MAX];
    for ( i = 0; i < MAX; i++)
      {
```

```
    ptr[i] = &var[i]; /* assign the address of integer. */
          }
    for ( i = 0; i < MAX; i++)
     {
    printf("Value of var[%d] = %d\n", i, *ptr[i] );
  }
    return 0;
}
```

Output:

```
Value of var[0] = 10
Value of var[1] = 100
Value of var[2] = 200
```

Function with Pointer

As we know that we can create a pointer of any data type such as int, char, float, we can also create a pointer pointing to a function.

The code of a function always resides in memory, which means that the function has some address. We can get the address of memory by using the function pointer.

Example:

```
#include<stdio.h>
void swap (int *a, int *b);
int main()
{
 int m = 25;
 int n = 100;
 printf("m is %d, n is %d\n", m, n);
 swap(&m, &n);
 printf("m is %d, n is %d\n", m, n);
 return 0;
 }
void swap (int *a, int *b)
{
 int temp;
 temp = *a;
 *a = *b;
 *b = temp;
```

```
}
```

Output:

```
m is 25, n is 100
m is 100, n is 25
```

Pointers to Structures

We can define pointers to structures in the same way as you define pointer to any other variable

```
struct Books *struct_pointer;
```

To access members of a structure using pointers, we use the `->` operator.

Example:

```
#include <stdio.h>
struct person
{
  int age;
  float weight;
};

int main()
{
   struct person *personPtr, person1;
   personPtr = &person1;
   printf("Enter age: ");
   scanf("%d", &personPtr->age);
   printf("Enter weight: ");
   scanf("%f", &personPtr->weight);
   printf("Displaying:\n");
   printf("Age: %d\n", personPtr->age);
   printf("weight: %f", personPtr->weight);
   return 0;
}
```

Output:

```
Enter age: 36
Enter weight: 36
```

Displaying:
Age: 36
weight: 36.000000

Storage Classes

Storage classes are used to determine the lifetime, visibility, memory location, and initial value of a variable.

There are four types of storage classes in C

- Automatic
- External
- Static
- Register

Storage Classes	Storage Place	Default Value	Scope	Lifetime
auto	RAM	Garbage Value	Local	Within function
extern	RAM	Zero	Global	Till the end of the main program Maybe declared anywhere in the program
static	RAM	Zero	Local	Till the end of the main program,Retains between multiple functions call
register	Register	Garbage Value	Local	Within the function

Auto Storage Class

- The variables defined using auto storage class are called as local variables.
- Auto stands for automatic storage class.
- A variable is in auto storage class by default if it is not explicitly specified.
- The scope of an auto variable is limited with the particular block only.
- Once the control goes out of the block, the access is destroyed.

- This means only the block in which the auto variable is declared can access it.
- A keyword auto is used to define an auto storage class.
- By default, an auto variable contains a garbage value.

Example:

```
#include <stdio.h>
int main()
{
auto int a; //auto
char b;
float c;
printf("a=%d \t b=%c \t c=%f",a,b,c);
return 0;
}
```

Output: a=0 b=. c=0.000000

Extern Storage Class

- Extern stands for external storage class.
- Extern storage class is used when we have global functions or variables which are shared between two or more files.
- Keyword **extern** is used to declaring a global variable or function in another file to provide the reference of variable or function which have been already defined in the original file.
- The variables defined using an extern keyword are called as global variables.
- These variables are accessible throughout the program.

Example:

```
#include <stdio.h>
extern int i;
int main()
{
  printf("value of the external integer is = %d\n", i);
  return 0;
  }
```

Output:

```
/usr/bin/ld: cannot open output file a.out: Permission denied
collect2: error: ld returned 1 exit status
```

Example:

```
#include <stdio.h>
extern int i=9;
int main()
{
  printf("value of the external integer is = %d\n", i);
  return 0;
}
```

Output: value of the external integer is = 9

Static Storage Class

- Static variables are visible only to the function or the block in which they are defined.
- A same static variable can be declared many times but can be assigned at only one time.
- Default initial value of the static integral variable is 0 otherwise null.
- The visibility of the static variable is limited to the file in which it has declared.
- The keyword used to define static variable is static.

Example:

```
#include<stdio.h>
static char c;
static int i;
static float f;
static char s[100];
void main ()
{
printf("%d %d %f %s",c,i,f,s);
}
```

Output: 0 0 0.000000

Register Storage Class

- The keyword **register** is used to declare a register storage class.
- The variables declared using register storage class has lifespan throughout the program.
- It is similar to the auto storage class.
- The variable is limited to the particular block.

- The only difference is that the variables declared using register storage class are stored inside CPU registers instead of a memory.
- Register has faster access than that of the main memory.
- The variables declared using register storage class has no default value.(Garbage value)

Example:

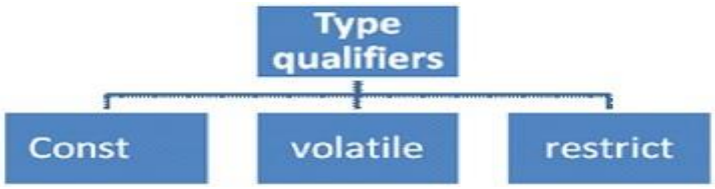
```
#include <stdio.h>
int main()
{
register int a;
printf("%d",a);
}
```

Output: 1059242368

Type qualifiers

Type qualifiers add special attributes to the existing datatypes.

There are three type qualifiers in C language there are Const, volatile and Restrict



Const

There are three types of constants, which are as follows –

- Literal constants
- Defined constants
- Memory constants

Literal constants: These are the unnamed constants that are used as

Example: a=b+7 //Here '7' is literal constant.

Defined constants: These constants use the preprocessor command 'define" with #

Example: #define PI 3.1415

Memory constants : These constants use 'C' qualifier 'const', which indicates that the data cannot be changed.

Example: const float pi = 3.1415

Example:

```
#include<stdio.h>
#define PI 3.1415
int main ( )
{
  const float cpi = 3.14;
  printf ("literal constant = %f",3.14);
  printf ("\n defined constant = %f", PI);
  printf ("\n memory constant = %f",cpi);
}
```

Output:

```
literal constant = 3.140000
 defined constant = 3.141500
 memory constant = 3.140000
```

Volatile

- A volatile keyword in C is nothing but a qualifier that is used by the programmer when they declare a variable in source code.
- It is used to inform the compiler that the variable value can be changed any time without any task given by the source code.
- Volatile is usually applied to a variable when we are declaring it.
- The main reason behind using volatile keyword is that it is used to prevent optimizations on objects in our source code.
- Therefore, an object declared as volatile can't be optimized because its value can be easily changed by the code.

Example:

```
#include<stdio.h>
volatile int a ;
int main()
{
a = 0 ;
if (a == 0)
{
printf ( " a = 0 \n " ) ;
}
else
{
printf ( " a ! = 0  \n " ) ;
}
```

```
return 0 ;
}
```
Output: a = 0

Restrict

- restrict keyword is mainly used in pointer declarations as a type qualifier for pointers.
- When we use restrict with a pointer ptr, it tells the compiler that ptr is the only way to access the object pointed by it and compiler doesn't need to add any additional checks.

Example:

```
#include <stdio.h>
void my_function(int* x, int* y, int* restrict z)
 {
   *x += *z;
   *y += *z;
}
main(void)
{
   int x = 10, y = 20, z = 30;
   my_function(&x, &y, &z);
   printf("%d %d %d", x, y, z);
}
```
Output: 40 50 30

Jumping Statements

Jumping statements are used to interrupt the normal flow of program.

Types of Jumping Statements

- Break
- Continue
- GoTo

Break Statement

- The break statement is used inside loop or switch statement.

- When compiler finds the break statement inside a loop, compiler will abort the loop and continue to execute statements followed by loop.

Example:

```
#include<stdio.h>
 void main()
     {
          int a=1;
          while(a<=10)
             {
              if(a==5)
               break;
              printf("\n Statement %d.",a);
                a++;
                          }
              printf("\nEnd of Program.");
     }
```

Output:

```
Statement 1.
Statement 2.
Statement 3.
Statement 4.
End of Program.
```

Continue Statement

- The continue statement is also used inside loop.
- When compiler finds the break statement inside a loop, compiler will skip all the followling statements in the loop and resume the loop.

Example:

```
#include<stdio.h>
     void main()
     {
          int a=0;
          while(a<5)
            {
              a++;
              if(a==3)
               continue;
               printf("\nStatement %d.",a);
                    }
        printf("\nEnd of Program.");
```

```
}
```

Output:

Statement 1.
Statement 2.
Statement 4.
Statement 5.
End of Program.

Goto Statement:

The goto statement is a jump statement which jumps from one point to another point within a function.

Example:

```
#include<stdio.h>

void main()
{
        printf("\nStatement 1.");
        printf("\nStatement 2.");
        printf("\nStatement 3.");

        goto last;

        printf("\nStatement 4.");
        printf("\nStatement 5.");

        last:

        printf("\nEnd of Program.");
}
```

Output:

Statement 1.
Statement 2.
Statement 3.
End of Program.

Command Line Arguments in C

- The arguments passed from command line are called command line arguments.

- These arguments are handled by main() function.
- To support command line argument, we need to change the structure of main() function as given below.
- int main(int argc, char *argv[])
- Here, argc counts the number of arguments. It counts the file name as the first argument.
- The argv[] contains the total number of arguments. The first argument is the file name always.

Example:

```
#include <stdio.h>
void main(int argc, char *argv[] )
  {
  printf("Program name is: %s\n", argv[0]);
     if(argc < 2)
       {
    printf("No argument passed through command line.\n");
             }
  else
     {
     printf("First argument is: %s\n", argv[1]);
            }
      }
```

Output:

```
Program name is: /tmp/N4r8W0ynUH.o
No argument passed through command line.
```