Module 4

1. Memory Allocation for Variables

Memory allocation for variables refers to the process by which a computer's memory is assigned to store the values of variables in a program. There are two primary types of memory allocation for variables: **static memory allocation** and **dynamic memory allocation**.

1.1. Static memory allocation:

If we are aware of the size of an array, then it is easy and we can define it as an array. For example, if we need to store the name of a person, then we can define an array to hold a maximum of 100 characters. We can define an array as follows –

```
char name[100];
```

1.2. Dynamic memory allocation:

Dynamic memory allocation allows the programmer to request memory during runtime. The key functions used in C for dynamic memory allocation are:

- **The malloc() Function**

  This function is defined in the **stdlib.h** header file. It allocates a block memory of the required size and returns a void pointer.

  ```
  void *malloc (size)
  ```

  The size parameter refers to the block of memory in bytes. To allocate a memory required for a specified data type, you need to use the typecasting operator.

  For example, the following snippet allocates the memory required to store an int type −

  ```
  int *ptr;

  ptr = (int *) malloc (sizeof (int));
  ```

Here we need to define a pointer to character without defining how much memory is required and later, based on requirement, we can allocate memory.

- **The calloc() Function**

  The calloc() function (stands for contiguous allocation) allocates the requested memory and returns a pointer to it.

  ```
  void *calloc(n, size);
  ```

Here, "n" is the number of elements to be allocated and "size" is the byte-size of each element.

The following snippet allocates the memory required to store 10 **int** types −

```
int *ptr;

ptr = (int *) calloc(10, sizeof(int));
```

- **The free() Function**

When your program comes out, the operating system automatically releases all the memory allocated by your program. However, it is a good practice to release the allocated memory explicitly by calling the **free()** function, when you are not in need of using the allocated memory anymore.

- **The realloc() Function**

The realloc() (re-allocation) function in C is used to dynamically change the memory allocation of a previously allocated memory. You can increase or decrease the size of an allocated memory block by calling the realloc() function.

The prototype of using the realloc() function is like this −

```
void *realloc(*ptr, size);
```

Here, the first parameter "**ptr**" is the pointer to a memory block previously allocated with malloc, calloc or realloc to be reallocated. If this is NULL, a new block is allocated and a pointer to it is returned by the function.

The second parameter "**size**" is the new size for the memory block, in bytes. If it is "0" and **ptr** points to an existing block of memory, the memory block pointed by **ptr** is deallocated and a NULL pointer is returned.

2. Pointers in C

Pointers A pointer is a derived data type in C. It is built from one of the fundamental data types available in C. Pointers contain memory addresses as their values. Since the data types addresses are the locations in the computer memory where program instructions and data are stored, pointers can be used to access and manipulate data stored in the memory.

Pointers are used frequently in C, as they offer a number of benefits to the programmers. They include:

1. Pointers are more efficient in handling arrays and data tables.
2. Pointers can be used to return multiple values from a function via function arguments.

3. Pointers permit references to functions and thereby facilitating passing of functions as arguments to other functions.
4. The use of pointer arrays to character strings results in saving of data storage space in memory.
5. Pointers allow C to support dynamic memory management
6. Pointers provide an efficient tool for manipulating dynamic data structures such as structures, linked lists, queues, stacks and trees.
7. Pointers reduce length and complexity of programs.
8. They increase the execution speed and thus reduce the program execution time.

2.1. Understanding Pointers

The computer's memory is a sequential collection of storage cells. Each cell, commonly known as a byte, has a number called address associated with it. Typically, the addresses are numbered consecutively, starting from zero. The last address depends on the memory size. A computer system having 64 K memory will have its last address 65,535.

Whenever we declare a variable, the system allocates, somewhere in the memory, an appropriate location to hold the value of the variable. Since, every byte has a unique address number, this location will have its own address number. Consider the following statement

int quantity = 179;

This statement instructs the system to find a location for the integer variable quantity and puts the value 179 in that location. Let us assume that the system has chosen the address location 5000 for quantity. We may represent this as shown in Fig. 2.1.

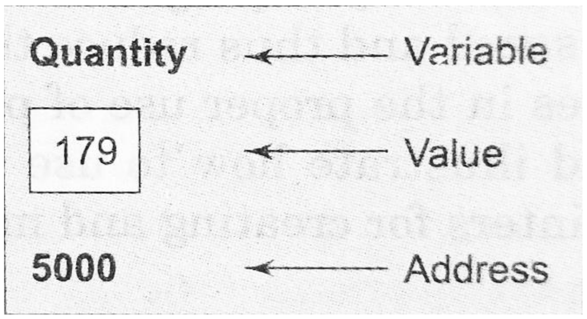


Figure 2.1. Representation of variable

During execution of the program, the system always associates the name quantity with the address 5000. We may have access to the value 179 by using either the name quantity or the address 5000. Since memory addresses are simply numbers, they can be assigned to some variables, that can be stored in memory, like any other variable. Such variables that hold memory addresses are called pointer variables. A pointer variable is, therefore, nothing but a variable that contains an address, which is a location of another variable in memory. Suppose, we assign the address of quantity to a variable p. The link between the variables p and quantity can be visualized as shown in Fig.2.2. The address of p is 5048.

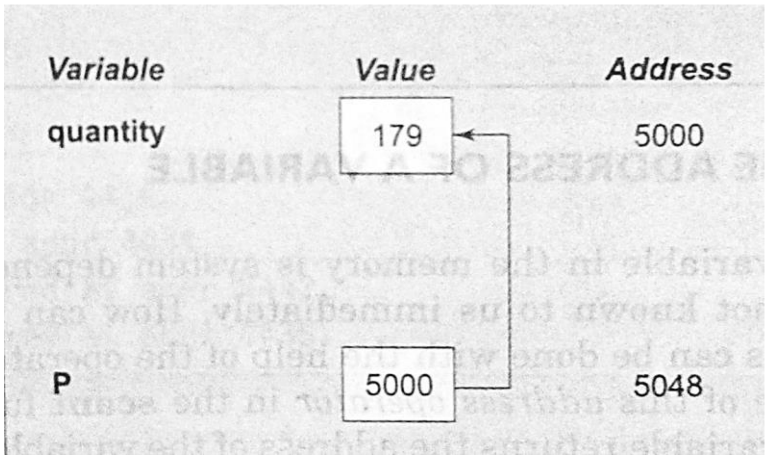
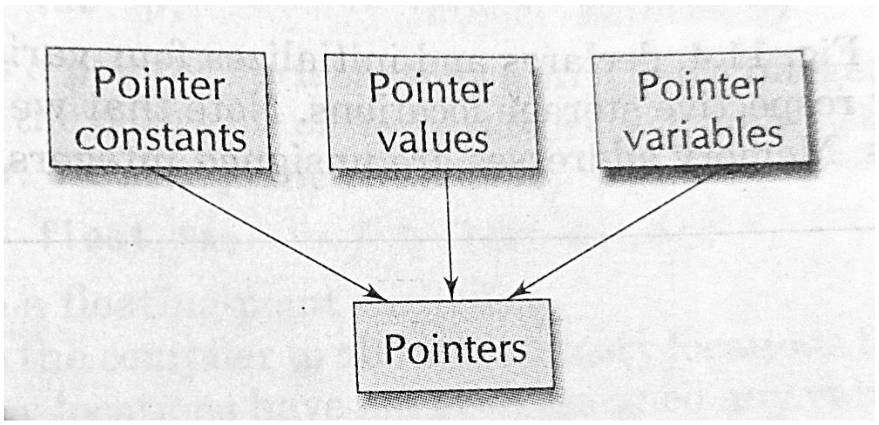


Figure 2.2. Pointer variable representation

Since the value of the variable p is the address of the variable quantity, we may access the value of quantity by using the value of p and therefore, we say that the variable p 'points' to the variable quantity. Thus, p gets the name 'pointer'.

2.1.1. Underlying Concepts of Pointers

Pointers are built on the three underlying concepts as illustrated below:



Memory addresses within a computer are referred to as **pointer constants**. We cannot change them; we can only use them to store data values. They are like house numbers.

We cannot save the value of a memory address directly. We can only obtain the value through the variable stored there using the address operator (&). The value thus obtained is known as **pointer value**. The pointer value (i.e. the address of a variable) may change from one run of the program to another.

Once we have a pointer value, it can be stored into another variable. The variable that contains a pointer value is called a **pointer variable**.

2.2. Accessing the Address of a Variable

The actual location of a variable in the memory is system dependent and therefore, the address of a variable is not known to us immediately. The address of a variable can be determined with the help of the operator ‘&’. The operator ‘&’ immediately preceding a variable returns the address of the variable associated with it. For example, the statement

P = &quantity;

would assign the address 5000 (the location of quantity) to the variable p. The & operator can be remembered as 'address of’. The & operator can be used only with a simple variable or an array element. The following are illegal use of address operator:

1. &125 (pointing at constants).
2. int x[10];
   &x (pointing at array names).
3. &(x+y) (pointing at expressions).

If x is an array, then expressions such as &x[0] and &x[i+3] are valid and represent the addresses of 0th and (i+3)th elements of x.

Example 2.3. A program to print the address of a variable along with its value.

The program below declares and initializes four variables and then prints out these values with their respective storage locations. Note that we have used %u format for printing address values. Memory addresses are unsigned integers.

Program

```
main()

{

char a;

int x;

float p, q;

a= 'A' ;

x = 125;

p = 10.25 , q = 18.76;

printf("%c is stored at addr %u . /n", a, &a) ;

printf("%d is stored at addr %u . /n", x, &x) ;

printf("%f is stored at addr % u . /n", p, &p) ;

printf("%f is stored at addr %u . /n", q, &q);
}
```

Output

```
A is stored at addr 4436.

125 is stored at addr 4434.

10.250000 is stored at addr 4442.

18.760000 is stored at addr 4438.
```

2.3. Declaring Pointer Variables

In C, every variable must be declared for its type. Since pointer variables contain addresses that belong to a separate data type, they must be declared as pointers before we use them. The declaration of a pointer variable takes the following form:

```
data_type *pt_name;
```

This tells the compiler three things about the variable pt_name.

1. The asterisk (*) tells that the variable pt_name is a pointer variable.
2. pt_name needs a memory location.
3. pt_name points to a variable of type data_type.

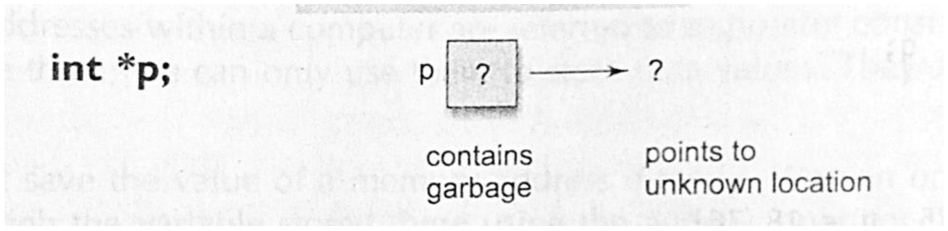
For example,

```
int *p;          /* integer pointer */
```

declares the variable p as a pointer variable that points to an integer data type. Remember that the type int refers to the data type of the variable being pointed to by p and not the type of the value of the pointer. Similarly, the statement

```
float *x;          /* float pointer */
```

declares x as a pointer to a floating-point variable. The declarations cause the compiler to allocate memory locations for the pointer variables p and x. Since the memory locations have not been assigned any values, these locations may contain some unknown values in them and therefore they point to unknown locations as shown:



2.3.1. Pointer Declaration Style

Pointer variables are declared similarly as normal variables except for the addition of the unary operator. This symbol can appear anywhere between the type name and the pointer variable name. Programmers use the following styles:

```
int*   p;      /* style 1 */
int   *p;      /* style 2 */
int    *   p;  /* style 3 */
```

2.4. Initialization of Pointer Variables

The process of assigning the address of a variable to a pointer variable is known as initialization. All uninitialized pointers will have some unknown values that will be interpreted as memory addresses. They may not be valid addresses, or they may point to some values that are wrong. Since the compilers do not detect these errors, the programs with uninitialized pointers will produce erroneous results. It is therefore important to initialize pointer variables carefully before they are used in the program.

Once a pointer variable has been declared we can use the assignment operator to initialize the variable. Example:

```
int quantity;
int *p;              /* declaration */
p = &quantity;       /* Initialization */
```

We can also combine the initialization with the declaration That is,

```
int *p = &quantity;
```

is allowed. The only requirement here is that the variable quantity must be declared before the initialization takes place. Remember, this is an initialization of p and not *p. Also, ensure that the pointer variables always point to the corresponding type of data as compiler will not detect such errors. For example,

```
float a, b;
int x, *p;
p = &a;        /* wrong */
b =*p;
```

will result in erroneous output because we are trying to assign the address of a float variable to an integer pointer.

It is also possible to combine the declaration of data variable, the declaration of pointer variable and the initialization of the pointer variable in one step. For example,

```
int x, * p = &x;            /* three in one */
```

is perfectly valid. It declares x as an integer variable and p as a pointer variable and then initializes p to the address of x.

The following statement is not valid.

```
int *p = &x, x;
```

We could also define a pointer variable with an initial value of NULL or 0 (zero). That is, the following statements are valued

```
int *p = NULL; int * p = 0 ;
```

With the exception of NULL and 0, no other constant value can be assigned to a pointer variable. For example, the following is wrong:

```
int *p = 5360;            /*absolute address */
```

2.5. Accessing a Variable Through Its Pointer

Once a pointer has been assigned the address of a variable, the value of the variable can be accessed through its pointer using another unary operator (asterisk), usually known as the indirection operator. Another name for the indirection operator is the dereferencing operator. Consider the following statements:

```
int quantity, *p, n;

quantity = 179;

p = &quantity:

n = * p;
```

The first line declares quantity and n as integer variables and p as a pointer variable pointing to an integer. The second line assigns the value 179 to quantity and the third line assigns the address of quantity to the pointer variable p. The fourth line contains the indirection operator *. When the operator * is placed before a pointer variable in an expression (on the right-hand side of the equal sign), the pointer returns the value of the variable of which the pointer value is the address. In this case, *p returns the value of the variable quantity, because p is the address of quantity. The * can be remembered as 'value at address'. Thus, the value of n would be 179. The two statements.

```
p = &quantity;

n = * p;
```

are equivalent to

```
n = *&quantity;
```

which in turn is equivalent to

n = quantity;

In C, the assignment of pointers and addresses is always done symbolically, by means of symbolic names. So, we cannot access the value stored at the address 5368 by writing *5368.

Example 2.5 Program to illustrate the use of indirection operator to access the value pointed to by a pointer. The value of pointer is 4104 and the value it points to is 10. Also, note the following equivalences:

x = *(&x) = *ptr = y

&x = &*ptr

Program

```
main()

{

int x, y;

int *ptr;

x= 10;

ptr = &x;

y= *ptr;

printf("Value of x is %d\n\n",x);

printf("%d is stored at addr %u\n", x, &x);

printf("%d is stored at addr %u\n", &x, &x);

printf("%d is stored at addr %u\n", *ptr, ptr); printf("%d is stored
at addr %u\n", ptr, &ptr);

printf("%d is stored at addr %u\n", y, &y);

*ptr = 25;

printf("\nNow x = %d\n",x);
```

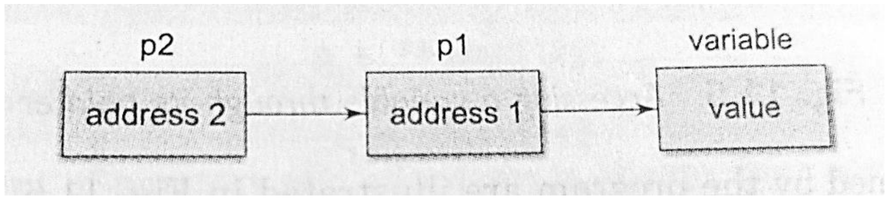
Output

```
Value of x is 10

10 is stored at addr 4104

10 is stored at addr 4104
```

```
10 is stored at addr 4104

4104 is stored at addr 4106

10 is stored at addr 4108

Now x = 25
```

2.6. Chain of Pointers

It is possible to make a pointer to point to another pointer, thus creating a chain of pointers as shown.



Here, the pointer variable p2 contains the address of the pointer variable p1, which points to the location that contains the desired value. This is known as multiple indirections. A variable that is a pointer to a pointer must be declared using additional indirection operator symbols in front of the name. Example:

```
int **p2;
```

This declaration tells the compiler that p2 is a pointer to a pointer of int type. Remember. the pointer p2 is not a pointer to an integer, but rather a pointer to an integer pointer and the target value can be accessed by applying the indirection operator twice. Consider the following code:

```
main ()

{

int x, *p1, **p2;

x =100;

p1 = &x;      /* address of x */

p2 = &p1;   /* address of p1 */

printf ("%d", **p2) ;

}
```

This code will display the value 100. Here, p1 is declared as a pointer to an integer and p2 as a pointer to a pointer to an integer.

2.7. Pointer Arithmetic

Pointer variables can also be used in arithmetic expressions. The pointer variables store the memory address of another variable. It doesn't store any value.

Hence, there are only a few operations that are allowed to perform on pointers in C language. The C pointer arithmetic operations are slightly different from the ones that we generally use for mathematical calculations. Following arithmetic operations are possible on the pointer in C language:

- Increment/Decrement
- Addition
- Subtraction
- Comparison

Increment: It is a condition that also comes under addition. When a pointer is incremented, it actually increments by the number equal to the size of the data type for which it is a pointer.

The rule to increment the pointer is given below:

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

For Example:

If an integer pointer that stores address 1000 is incremented, then it will increment by 4(size of an int), and the new address will point to 1004. While if a float type pointer is incremented then it will increment by 4(size of a float) and the new address will be 1004.

Demo Program:

```
#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 increment: Address of p variable is %u \n",p); // in our
case, p will get incremented by 4 bytes.
return 0;
}
```

Output

```
Address of p variable is 3214864300
```

```
After increment: Address of p variable is 3214864304
```

Decrement: It is a condition that also comes under subtraction. When a pointer is decremented, it actually decrements by the number equal to the size of the data type for which it is a pointer.

The formula for decrementing the pointer is given below:

*new_address= current_address - i * size_of(data type)*

For Example:

If an integer pointer that stores address 1000 is decremented, then it will decrement by 4(size of an int), and the new address will point to 996. While if a float type pointer is decremented then it will decrement by 4(size of a float) and the new address will be 996.

Demo Program:

```
#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 decrement: Address of p variable is %u \n",p); // P will
now point to the immidiate previous location.
return 0;
}
```

Output

```
Address of p variable is 3214864300
After decrement: Address of p variable is 3214864296
```

Pointer Addition: When a pointer is added with an integer value, the value is first multiplied by the size of the data type and then added to the pointer.

The formula of adding value to pointer is given below:

*new_address= current_address + (number * size_of(data type))*

For Example:

Consider the same example as above where the ptr is an integer pointer that stores 1000 as an address. If we add integer 5 to it using the expression, ptr = ptr + 5, then, the final address stored in the ptr will be ptr = 1000 + sizeof(int) * 5 = 1020.

Demo Program:

```
#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+3; //adding 3 to pointer variable
printf("After adding 3: Address of p variable is %u \n",p);
return 0;
}
```

Output

```
Address of p variable is 3214864300
After adding 3: Address of p variable is 3214864312
```

Pointer Subtraction:

a. *Subtraction of Integer to Pointer*

When a pointer is subtracted with an integer value, the value is first multiplied by the size of the data type and then subtracted from the pointer similar to addition.

The formula of subtracting value from the pointer variable is given below:

*new_address= current_address - (number * size_of(data type))*

For Example:

Consider the same example as above where the ptr is an integer pointer that stores 1000 as an address. If we subtract integer 5 from it using the expression, ptr = ptr – 5, then, the final address stored in the ptr will be ptr = 1000 – sizeof(int) * 5 = 980.

Demo Program:

```
#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-3; //subtracting 3 from pointer variable
printf("After subtracting 3: Address of p variable is %u \n",p);
return 0;
}
```

Output

```
Address of p variable is 3214864300
After subtracting 3: Address of p variable is 3214864288
```

b. *Subtraction of Two Pointers*

Instead of subtracting a number, we can also subtract an address from another address (pointer). This will result in a number. The subtraction of two pointers gives the increments between the two pointers. It will not be a simple arithmetic operation, but it will follow the following rule.

If two pointers are of the same type,

Address2 - Address1 = (Subtraction of two addresses)/size of data type which pointer points

For Example:

Two integer pointers say ptr1(address:1000) and ptr2(address:1004) are subtracted. The difference between addresses is 4 bytes. Since the size of int is 4 bytes, therefore the increment between ptr1 and ptr2 is given by (4/4) = 1.

Demo Program:

```
#include <stdio.h>
// Driver Code
void main()
{
    int x = 6; // Integer variable declaration
    int N = 4;
    // Pointer declaration
    int *ptr1, *ptr2;
    ptr1 = &N; // stores address of N
    ptr2 = &x; // stores address of x
    printf(" ptr1 = %u, ptr2 = %u\n", ptr1, ptr2);
    // %p gives an hexa-decimal value,
   // We convert it into an unsigned int value by using %u
   // Subtraction of ptr2 and ptr1
    x = ptr1 - ptr2;
    // Print x to get the Increment between ptr1 and ptr2
    printf("Subtraction of ptr1 & ptr2 is %d\n",x);
}
```

Output

```
ptr1 = 2715594428, ptr2 = 2715594424
Subtraction of ptr1 & ptr2 is 1
```

Pointer Comparison: We can compare the two pointers by using the comparison operators in C. This can be implemented by using all operators in C >, >=, <, <=, ==, !=. It returns true for the valid condition and returns false for the unsatisfied condition.

Step 1: Initialize the integer values and point these integer values to the pointer.

Step 2: Now, check the condition by using comparison or relational operators on pointer variables.

Step 3: Display the output.

Demo Program:

```
#include <stdio.h>
void main()
{
    // declaring array
    int arr[5];
    // declaring pointer to array name
    int* ptr1 = &arr;
    // declaring pointer to first element
 S   int* ptr2 = &arr[0];
    if (ptr1 == ptr2) {
        printf("Pointer to Array Name and First Element "
               "are Equal.");
    }
    else {
        printf("Pointer to Array Name and First Element "
               "are not Equal.");
    }
}
```

Output

```
Pointer to Array Name and First Element are Equal.
```

3. Structures

The structure in C is a user-defined data type that can be used to group items of possibly different types into a single type. The struct keyword is used to define the structure in the C programming language. The items in the structure are called its member and they can be of any valid data type. Additionally, the values of a structure are stored in contiguous memory locations.

3.1. C Structure Declaration

We have to declare structure in C before using it in our program. In structure declaration, we specify its member variables along with their datatype. We can use the struct keyword to declare the structure in C using the following syntax:

Syntax

```
struct structure_name {
   data_type member_name1;
   data_type member_name1;
   ....
   ....
};
```

The above syntax is also called a structure template or structure prototype and no memory is allocated to the structure in the declaration.

3.2. C Structure Definition

To use structure in our program, we have to define its instance. We can do that by creating variables of the structure type. We can define structure variables using two methods:

1. Structure Variable Declaration with Structure Template

```
struct structure_name {
   data_type member_name1;
   data_type member_name1;
```

```
                ....

                ....

            }variable1, varaible2, ...;
```

2. Structure Variable Declaration after Structure Template

```
            // structure declared beforehand

            struct structure_name variable1, variable2, .......;
```

3.2.1. Access Structure Members

We can access structure members by using the (.) dot operator.

<u>*Syntax*</u>

```
    structure_name.member1;

    strcuture_name.member2;
```

In the case where we have a pointer to the structure, we can also use the arrow operator to access the members.

3.2.2. Initialize Structure Members

Structure members cannot be initialized with the declaration. For example, the following C program fails in the compilation.

```
    struct Point

    {

       int x = 0;  // COMPILER ERROR:  cannot initialize members here

       int y = 0;  // COMPILER ERROR:  cannot initialize members here

    };
```

The reason for the above error is simple. When a datatype is declared, no memory is allocated for it. Memory is allocated only when variables are created.

Default Initialization

By default, structure members are not automatically initialized to 0 or NULL. Uninitialized structure members will contain garbage values. However, when a structure variable is declared with an initializer, all members not explicitly initialized are zero-initialized.

```
struct Point
{
    int x;
    int y;
};
```

struct Point p = {0}; // Both x and y are initialized to 0

We can initialize structure members in 3 ways which are as follows:

i. Using Assignment Operator.
ii. Using Initializer List.
iii. Using Designated Initializer List.

i. Initialization using Assignment Operator

```
struct structure_name str;
str.member1 = value1;
str.member2 = value2;
str.member3 = value3;
```

ii. Initialization using Initializer List

```
struct structure_name str = { value1, value2, value3 };
```

In this type of initialization, the values are assigned in sequential order as they are declared in the structure template.

iii. Initialization using Designated Initializer List

Designated Initialization allows structure members to be initialized in any order. This feature has been added in the C99 standard.

```
struct structure_name str = { .member1 = value1, .member2 = value2, .member3 = value3 };
```

The Designated Initialization is only supported in C but not in C++.

Example of Structure in C

The following C program shows how to use structures

Program

```
#include <stdio.h>
// declaring structure with name str1
struct str1 {
    int i;
    char c;
    float f;
    char s[30];
};
// declaring structure with name str2
struct str2 {
    int ii;
    char cc;
    float ff;
} var; // variable declaration with structure template

// Driver code
int main()
{
    // variable declaration after structure template
    // initialization with initializer list and designated
    //    initializer list
    struct str1 var1 = { 1, 'A', 1.00, "GeeksforGeeks" }, var2;
    struct str2 var3 = { .ff = 5.00, .ii = 5, .cc = 'a' };

    // copying structure using assignment operator
    var2 = var1;

    printf("Struct 1:\n\ti = %d, c = %c, f = %f, s = %s\n",
            var1.i, var1.c, var1.f, var1.s);
    printf("Struct 2:\n\ti = %d, c = %c, f = %f, s = %s\n",
            var2.i, var2.c, var2.f, var2.s);
    printf("Struct 3\n\ti = %d, c = %c, f = %f\n", var3.ii,
            var3.cc, var3.ff);
    return 0;
}
```

Output

```
Struct 1:
  i = 1, c = A, f = 1.000000, s = GeeksforGeeks
Struct 2:
  i = 1, c = A, f = 1.000000, s = GeeksforGeeks
Struct 3
  i = 5, c = a, f = 5.000000
```

3.3. typedef for Structures

The typedef keyword is used to define an alias for the already existing datatype. In structures, we have to use the struct keyword along with the structure name to define the variables. Sometimes, this increases the length and complexity of the code. We can use the typedef to define some new shorter name for the structure.

Example Program

```
// C Program to illustrate the use of typedef with
// structures
#include <stdio.h>
// defining structure
typedef struct {
    int a;
} str1;

// another way of using typedef with structures
typedef struct {
    int x;
} str2;

int main()
{
    // creating structure variables using new names
    str1 var1 = { 20 };
    str2 var2 = { 314 };

    printf("var1.a = %d\n", var1.a);
    printf("var2.x = %d\n", var2.x);

    return 0;
}
```

Output

```
var1.a = 20

var2.x = 314
```

3.4. Nested Structures

C language allows us to insert one structure into another as a member. This process is called nesting and such structures are called nested structures. There are two ways in which we can nest one structure into another:

a. Embedded Structure Nesting

In this method, the structure being nested is also declared inside the parent structure.

Example

```
struct parent {
   int member1;
   struct member_str member2 {
      int member_str1;
      char member_str2;
      ...
   }
   ...
}
```

b. Separate Structure Nesting

In this method, two structures are declared separately and then the member structure is nested inside the parent structure.

Example

```
struct member_str {
   int member_str1;
   char member_str2;
   ...
}

struct parent {
   int member1;
   struct member_str member2;
   ...
}
```

One thing to note here is that the declaration of the structure should always be present before its definition as a structure member. For example, the declaration below is invalid as the struct mem is not defined when it is declared inside the parent structure.

```
struct parent {
    struct mem a;
};


struct mem {
    int var;
};
```

3.4.1. Accessing Nested Members

We can access nested Members by using the same (.) dot operator two times as shown:

```
str_parent.str_child.member;
```

Example Program of Structure Nesting

// C Program to illustrate structure nesting along with

// forward declaration

```
#include <stdio.h>
// child structure declaration
struct child {
    int x;
    char c;
};
// parent structure declaration
struct parent {
    int a;
    struct child b;
};
```

```
// driver code
int main()
{
    struct parent var1 = { 25, 195, 'A' };

    // accessing and printing nested members
    printf("var1.a = %d\n", var1.a);
    printf("var1.b.x = %d\n", var1.b.x);
    printf("var1.b.c = %c", var1.b.c);

    return 0;
}
```

Output

```
var1.a = 25

var1.b.x = 195

var1.b.c = A
```

4. Files in C

File handling in C is the process in which we create, open, read, write, and close operations on a file. C language provides different functions such as fopen(), fwrite(), fread(), fseek(), fprintf(), etc. to perform input, output, and many different C file operations in our program.

Advantages of using files in programming:

- **Reusability:** The data stored in the file can be accessed, updated, and deleted anywhere and anytime providing high reusability.
- **Portability:** Without losing any data, files can be transferred to another in the computer system. The risk of flawed coding is minimized with this feature.
- **Efficient:** A large amount of input may be required for some programs. File handling allows you to easily access a part of a file using few instructions which saves a lot of time and reduces the chance of errors.

- **Storage Capacity:** Files allow you to store a large amount of data without having to worry about storing everything simultaneously in a program.

4.1. Types of Files in C

A file can be classified into two types based on the way the file stores the data.

1. **Text Files :** A text file contains data in the form of ASCII characters and is generally used to store a stream of characters.
    - Each line in a text file ends with a new line character ('\n').
    - It can be read or written by any text editor.
    - They are generally stored with **.txt** file extension.
    - Text files can also be used to store the source code.
2. **Binary Files :** A binary file contains data in binary form (i.e. 0's and 1's) instead of ASCII characters. They contain data that is stored in a similar manner to how it is stored in the main memory.
    - The binary files can be created only from within a program and their contents can only be read by a program.
    - More secure as they are not easily readable.
    - They are generally stored with .bin file extension.

4.2. C File Operations

C file operations refer to the different possible operations that we can perform on a file in C such as:

a. Opening an existing file – **fopen()**

b. Creating a new file – **fopen()**

c. Reading from file – **fscanf() or fgets()**

d. Writing to a file – **fprintf() or fputs()**

e. Moving to a specific location in a file – **fseek(), rewind()**

f. Closing a file – **fclose()**

File Pointer in C

A file pointer is a reference to a particular position in the opened file. It is used in file handling to perform all file operations such as read, write, close, etc. We use the **FILE** macro to declare the file pointer variable. The FILE macro is defined inside **<stdio.h>** header file.

Syntax of File Pointer

FILE* *pointer_name*;

File Pointer is used in almost all the file operations in C.

a. <u>Open a File in C</u>

For opening a file in C, the fopen() function is used with the filename or file path along with the required access modes.

Syntax of fopen()

FILE* **fopen**(const char **file_name*, const char **access_mode*);

Parameters

- *file_name**:*** name of the file when present in the same directory as the source file. Otherwise, full path.
- *access_mode:* Specifies for what operation the file is being opened.

Return Value

- If the file is opened successfully, returns a file pointer to it.
- If the file is not opened, then returns NULL.

File opening modes in C

File opening modes or access modes specify the allowed operations on the file to be opened. They are passed as an argument to the fopen() function. Some of the commonly used file access modes are listed below

- **r** : Searches file. If the file is opened successfully fopen( ) loads it into memory and sets up a pointer that points to the first character in it. If the file cannot be opened fopen( ) returns NULL.
- **rb** : Open for reading in binary mode. If the file does not exist, fopen( ) returns NULL.
- w : Open for writing in text mode. If the file exists, its contents are overwritten. If the file doesn't exist, a new file is created. Returns NULL, if unable to open the file.
- wb : Open for writing in binary mode. If the file exists, its contents are overwritten. If the file does not exist, it will be created.
- a : Searches file. If the file is opened successfully fopen( ) loads it into memory and sets up a pointer that points to the last character in it. It opens only in the append mode. If the file doesn't exist, a new file is created. Returns NULL, if unable to open the file.

- ab : Open for append in binary mode. Data is added to the end of the file. If the file does not exist, it will be created.
- r+ : Searches file. It is opened successfully fopen( ) loads it into memory and sets up a pointer that points to the first character in it. Returns NULL, if unable to open the file.
- rb+ : Open for both reading and writing in binary mode. If the file does not exist, fopen( ) returns NULL.
- w+ : Searches file. If the file exists, its contents are overwritten. If the file doesn't exist a new file is created. Returns NULL, if unable to open the file.
- wb+: Open for both reading and writing in binary mode. If the file exists, its contents are overwritten. If the file does not exist, it will be created.
- a+ : Searches file. If the file is opened successfully fopen( ) loads it into memory and sets up a pointer that points to the last character in it. It opens the file in both reading and append mode. If the file doesn't exist, a new file is created. Returns NULL, if unable to open the file.
- ab+ : Open for both reading and appending in binary mode. If the file does not exist, it will be created

Example :C Program to illustrate file opening

```
#include <stdio.h>

#include <stdlib.h>

int main()

{

   FILE* fptr;

   fptr = fopen("filename.txt", "r");

   if (fptr == NULL)

   {

      printf("The file is not opened. The program will "

           "now exit.");

      exit(0);

   }

 return 0;
}
```

b. <u>Create a File in C</u>

The fopen() function can not only open a file but also can create a file if it does not exist already. For that, we have to use the modes that allow the creation of a file if not found such as w, w+, wb, wb+, a, a+, ab, and ab+.

FILE *fptr;
fptr = **fopen**("*filename.txt*", "**w**");

<u>Example: C Program to create a file</u>

```
#include <stdio.h>

#include <stdlib.h>

int main()

{

   FILE* fptr;

   fptr = fopen("file.txt", "w");

   if (fptr == NULL)

    {

      printf("The file is not opened. The program will exit now");

      exit(0);

   }

   else

   {

      printf("The file is created Successfully.");

   }

 return 0;

}
```

c. <u>**Reading From a File**</u>

The file read operation in C can be performed using functions fscanf() or fgets(). Both the functions performed the same operations as that of scanf and gets but with an additional parameter, the file pointer. There are also other functions we can use to read from a file. Such functions are listed below:

- fscanf() : Use formatted string and variable arguments list to take input from a file.
- fgets(): Input the whole line from the file.
- fgetc() : Reads a single character from the file.
- fgetw() : Reads a number from a file.
- fread() : Reads the specified bytes of data from a binary file.

Example:

```
FILE * fptr;
fptr = fopen(“fileName.txt”, “r”);
fscanf(fptr, "%s %s %s %d", str1, str2, str3, &year);
char c = fgetc(fptr);
```

The getc() and some other file reading functions return EOF (End Of File) when they reach the end of the file while reading. EOF indicates the end of the file and its value is implementation-defined.

d. <u>**Write to a File**</u>

The file write operations can be performed by the functions fprintf() and fputs() with similarities to read operations. C programming also provides some other functions that can be used to write data to a file such as:

- fprintf() : Similar to printf(), this function use formatted string and varible arguments list to print output to the file
- fputs(): Prints the whole line in the file and a newline at the end.
- fputc(): Prints a single character into the file.
- fputw(): Prints a number to the file.
- fwrtite(): This functions write the specified amount of bytes to the binary file.

<u>Example:</u>

```
FILE *fptr ;
fptr = fopen(“fileName.txt”, “w”);
fprintf(fptr, "%s %s %s %d", "We", "are", "in", 2012);
fputc("a", fptr);
```

e. <u>Move to a specific location in a file</u>

Both C's **fseek() and rewind()** functions are used to position files but serve different purposes. **fseek()** is used to set the file position indicator to a specified offset from the specified origin.

<u>Syntax</u>

```
int fseek(FILE *stream, long offset, int origin);
```

<u>Parameters</u>

stream: It is a pointer to the FILE stream.
offset: It specifies the number of bytes to offset from the origin.
origin: It indicates the position from where the offset is calculated. It can be one of the following:
SEEK_SET: Beginning of the file.
SEEK_CUR: Current position of the file pointer.
SEEK_END: End of the file.

<u>Return Value</u>
It returns 0 if successful, and a non-zero value if an error occurs.

rewind() is used to move the file pointer to the beginning of the file stream.

<u>Syntax</u>

```
void rewind(FILE *stream);
```

<u>Parameters</u>

stream: It is the pointer to the file stream

<u>Return Value</u>
It does not return any value.

f. <u>Closing a File</u>

The fclose() function is used to close the file. After successful file operations, you must always close a file to remove it from the memory.

<u>Syntax</u>

fclose(*file_pointer*);

where the *file_pointer* is the pointer to the opened file.

<u>Example:</u>

```
FILE *fptr ;
fptr= fopen(“fileName.txt”, “w”);

 //Some file Operations //

fclose(fptr);
```

<u>File Handing in C</u>

<u>Example 1 : C program to Open a File, Write in it, And Close the File</u>

```
#include <stdio.h>

#include <string.h>

int main()

{

   FILE* filePointer;

   char dataToBeWritten[50] = "GeeksforGeeks-A Computer Science Portal for Geeks";

   filePointer = fopen("GfgTest.c", "w");

   if (filePointer == NULL)

   {

      printf("GfgTest.c file failed to open.");

    }

   else

   {

      printf("The file is now opened.\n");

      if (strlen(dataToBeWritten) > 0)

          {

              fputs(dataToBeWritten, filePointer);
```

```
                    fputs("\n", filePointer);
            }
         fclose(filePointer);
         printf("Data successfully written in file GfgTest.c\n");
        printf("The file is now closed.");
      }
  return 0;
}
```

<u>Example 2: Program to Open a File, Read from it, And Close the File</u>

```
#include <stdio.h>
#include <string.h>
int main()
{
     FILE* filePointer;
    char dataToBeRead[50];
    filePointer = fopen("GfgTest.c", "r");
   if (filePointer == NULL)
    {
      printf("GfgTest.c file failed to open.");
     }
   else
   {
      printf("The file is now opened.\n");
       while (fgets(dataToBeRead, 50, filePointer != NULL)
            {

             printf("%s", dataToBeRead);

            }

        fclose(filePointer);

        printf("Data successfully read from file GfgTest.c\n");

        printf("The file is now closed.");

     }

   return 0;

}
```

Example 3:

Here is an example program that uses C's fseek() function. In this program, a straightforward text file will be created, some data will be written into it, and then fseek() will be used to read and output data from a given location in the file.

```
#include <stdio.h>
int main() {
   FILE *file;
const char *filename = "example.txt";
   char buffer[100];
   // Open the file in write mode to create and write data
   file = fopen(filename, "w");
   if (file == NULL) {
perror("Error opening the file");
       return 1;
   }
   // Write some data into the file
fputs("Hello, this is a sample file.\n", file);
fputs("This is line 2.\n", file);
fputs("This is line 3.\n", file);
   // Close the file
fclose(file);
   // Open the file in read mode
   file = fopen(filename, "r");
   if (file == NULL) {
perror("Error opening the file");
```

```
        return 1;
    }
    // Move the file pointer to the start of the third line (skipping two lines)
fseek(file, 0, SEEK_SET); // Start from the beginning of the file
fgets(buffer, sizeof(buffer), file); // Read line 1
fgets(buffer, sizeof(buffer), file); // Read line 2
    // Now, the file pointer is at the start of line 3. Print it and subsequent lines.
    while (fgets(buffer, sizeof(buffer), file) != NULL) {
printf("%s", buffer);
    }
    // Close the file
fclose(file);
    return 0;
}
```

Output:

```
This is line 3.
```

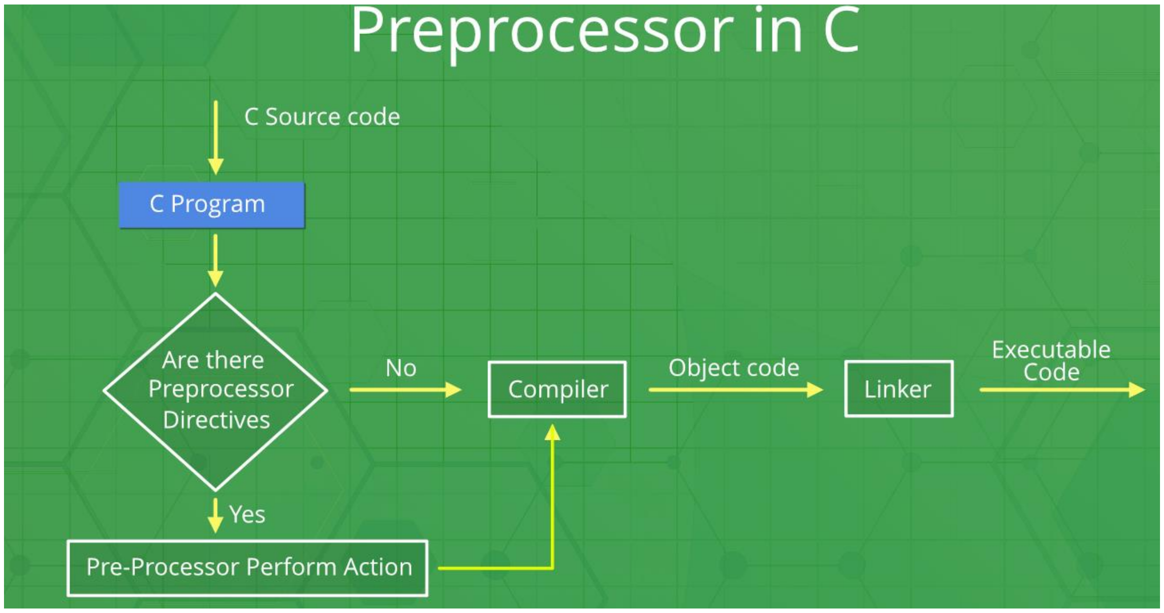
Example 4: This following code example sets the file position indicator of an input stream back to the beginning using rewind(). But, there is no way to check whether the rewind() was successful.

Program

```
int main()
{
    FILE* fp = fopen("test.txt", "r");

    if (fp == NULL) {
        /* Handle open error */
    }

    /* Do some processing with file*/

    /* no way to check if rewind is successful */
    rewind(fp);

    /* Do some more precessing with file */

    return 0;
}
```

5. C Preprocessors

Preprocessors are programs that process the source code before compilation Several steps are involved between writing a program and executing a program in C.



The source code written by programmers is first stored in a file, let the name be "**program.c**". This file is then processed by preprocessors and an expanded source code file is generated named "program.i". This expanded file is compiled by the compiler and an object code file is generated named "program.obj". Finally, the linker links this object code file to the object code of the library functions to generate the executable file "program.exe".

5.1. Preprocessor directives

Preprocessor programs provide preprocessor directives that tell the compiler to preprocess the source code before compiling. All of these preprocessor directives begin with a '#' (hash) symbol. The '#' symbol indicates that whatever statement starts with a '#' will go to the preprocessor program to get executed.

Examples of some preprocessor directives are: *#include*, *#define*, *#ifndef,* etc.

List of preprocessor directives in C:

Preprocessor Directives	Description
#define	Used to define a macro
#undef	Used to undefine a macro
#include	Used to include a file in the source code program
#ifdef	Used to include a section of code if a certain macro is defined by #define
#ifndef	Used to include a section of code if a certain macro is not defined by #define
#if	Check for the specified condition
#else	Alternate code that executes when #if fails
#elif	Combines else and if for another condition check
#endif	Used to mark the end of #if, #ifdef, and #ifndef

5.2. Types of C Preprocessors

There are 4 Main Types of Preprocessor Directives:

- Macros
- File Inclusion
- Conditional Compilation
- Other directives

1. Macros

In C, Macros are pieces of code in a program that is given some name. Whenever this name is encountered by the compiler, the compiler replaces the name with the actual piece of code. The **‘#define’** directive is used to define a macro.

Syntax of Macro Definition

```
#define token value
```

where after preprocessing, the *token* will be expanded to its *value* in the program.

```
// C Program to illustrate the macro
#include <stdio.h>
// macro definition
#define LIMIT 5
int main()
{
   for (int i = 0; i < LIMIT; i++) {
      printf("%d \n", i);
   }
   return 0;
}
```

Output

```
0

1

2

3

4
```

In the above program, when the compiler executes the word LIMIT, it replaces it with 5. The word **'LIMIT'** in the macro definition **is called a macro template** and **'5' is macro expansion.**

Macros With Arguments

We can also pass arguments to macros. Macros defined with arguments work similarly to functions.

```
#define foo(a, b) a + b
#define func(r) r * r
```

Example

```
// C Program to illustrate function like macros
#include <stdio.h>
#define AREA(l, b) (l * b)
int main()
{
   int l1 = 10, l2 = 5, area;
   area = AREA(l1, l2);
```

```
    printf("Area of rectangle is: %d", area);
    return 0;
}
```
Output
```
Area of rectangle is: 50
```

In the above program that whenever the compiler finds AREA(l, b) in the program, it replaces it with the statement (l*b). Not only this, but the values passed to the macro template AREA(l, b) will also be replaced in the statement (l*b). Therefore AREA(10, 5) will be equal to 10*5.

2. File Inclusion

This type of preprocessor directive tells the compiler to include a file in the source code program. The **#include preprocessor directive** is used to include the header files in the C program.

There are two types of files that can be included by the user in the program:

1. **Standard Header Files**

The standard header files contain definitions of pre-defined functions like **printf(), scanf(),** etc. These files must be included to work with these functions. Different functions are declared in different header files.

For example, standard I/O functions are in the 'iostream' file whereas functions that perform string operations are in the 'string' file.
Syntax
#include <*file_name*>
where *file_name* is the name of the header file to be included. The **'<' and '>' brackets** tell the compiler to look for the file in the s**tandard directory.**

2. **User-defined Header Files**

When a program becomes very large, it is a good practice to divide it into smaller files and include them whenever needed. These types of files are user-defined header files.

Syntax
#include "*filename*"

The **double quotes (" ")** tell the compiler to search for the header file in the **source file's directory.**

5.3. Preparing customized header files

We can create our own header files and include them in our program to use whenever we want. It enhances code functionality and readability. Both the file must be in the same folder. Below are the steps to create our own header file:

Step 1: Write your own C code and save that file with the **".h"** extension. Eg. myhead.h

```
void add(int a, int b)
{
        printf("Added value=%d\n", a + b);
}
void multiply(int a, int b)
{
        printf("Multiplied value=%d\n", a * b);
}
```

Step 2: Include your header file with **"#include"** in your C program as shown below:

```
// C program to use the above created header file
#include <stdio.h>
#include "myhead.h"
int main()
{
        add(4, 6);        //This calls add function written in myhead.h and therefore no compilation
error.
        multiply(5, 5);   // Same for the multiply function in myhead.h
        printf("BYE!See you Soon");
        return 0;
}
```

Output:

```
Added value:10
Multiplied value:25
BYE!See you Soon
```