

Unit -5 .User defined functions, pointers, file management and preprocessor directives.

User-Defined Function in C

A **user-defined function** is a type of function in C language that is defined by the user himself to perform some specific task. It provides code reusability and modularity to our program.

To use a user-defined function, we first have to understand the different parts of its syntax. The user-defined function in C can be divided into three parts:

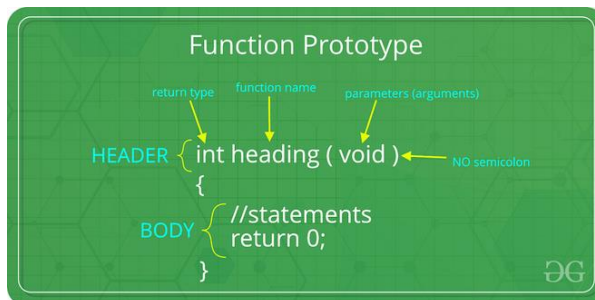
1. Function Prototype or declaration
2. Function Definition
3. Function Call

C Function Prototype or Function Declaration

A function prototype is also known as a function declaration which specifies the **function's name**, **function parameters**, and **return type**. The function prototype does not contain the body of the function. It is basically used to inform the compiler about the existence of the user-defined function which can be used in the later part of the program.

Syntax

`return_type function_name (type1 arg1, type2 arg2, ... typeN argN);`



C Function Definition

Once the function has been called, the function definition contains the actual statements that will be executed. All the statements of the function definition are enclosed within **{ } braces**.

Syntax

```
return_type function_name (type1 arg1, type2 arg2 .... typeN argN) {  
    // actual statements to be executed  
    // return value if any  
}
```

C Function Call

In order to transfer control to a user-defined function, we need to call it. Functions are called using their names followed by round brackets. Their arguments are passed inside the brackets.

Syntax

`function_name(arg1, arg2, ... argN);`

// C Program to illustrate the use of user-defined function

```
#include <stdio.h>  
int sum(int, int);    // Function prototype  
int sum(int x, int y) // Function definition  
{  
    int sum;  
    sum = x + y;  
    return x + y;  
}  
// Driver code  
int main()  
{  
    int x = 10, y = 11;  
    int result = sum(x, y); // Function call  
    printf("Sum of %d and %d = %d ", x, y, result);  
    return 0;  
}
```

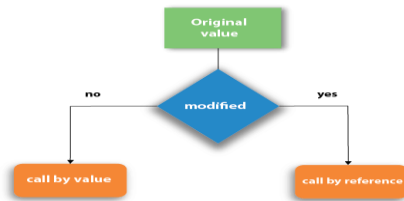
Output

Sum of 10 and 11 = 21

Passing Parameters to User-Defined Functions

We can pass parameters to a function in C using two methods:

1. Call by Value
2. Call by Reference



Call by value

- In call by value method, the value of the actual parameters is copied into the formal parameters. In other words, we can say that the value of the variable is used in the function call in the call by value method.
- In call by value method, we can not modify the value of the actual parameter by the formal parameter.
- In call by value, different memory is allocated for actual and formal parameters since the value of the actual parameter is copied into the formal parameter.
- The actual parameter is the argument which is used in the function call whereas formal parameter is the argument which is used in the function definition.

Call by Value Example: Swapping the values of the two variables

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

```
void swap(int , int); //prototype of the function
```

```
int main()
```

```
{
```

```
int a = 10;
```

```
int b = 20;
```

```
printf("Before swapping the values in main a = %d, b = %d\n",a,b); // printing the value of a and b in main
```

```
swap(a,b);
```

```
printf("After swapping values in main a = %d, b = %d\n",a,b); // The value of actual parameters do not change by changing the formal parameters in call by value, a = 10, b = 20
```

```
}
```

```
void swap (int a, int b)
```

```
{
```

```
int temp;
```

```
temp = a;
```

```
a=b;
```

```
b=temp;
```

```
printf("After swapping values in function a = %d, b = %d\n",a,b); // Formal parameters, a = 20, b = 10
```

```
}
```

Output

Before swapping the values in main a = 10, b = 20

After swapping values in function a = 20, b = 10

After swapping values in main a = 10, b = 20

Call by reference

- In call by reference, the address of the variable is passed into the function call as the actual parameter.
- The value of the actual parameters can be modified by changing the formal parameters since the address of the actual parameters is passed.
- In call by reference, the memory allocation is similar for both formal parameters and actual parameters. All the operations in the function are performed on the value stored at the address of the actual parameters, and the modified value gets stored at the same address.

Call by reference Example: Swapping the values of the two variables

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

```
void swap(int *, int *); //prototype of the function
```

```
int main()
```

```
{
```

```
int a = 10;
```

```
int b = 20;
```

```
printf("Before swapping the values in main a = %d, b = %d\n",a,b); // printing the value of a and b in main
```

```
swap(&a,&b);
```

```
printf("After swapping values in main a = %d, b = %d\n",a,b); // The values of actual parameters do change in call by reference, a = 10, b = 20
```

```
}
```

```
void swap (int *a, int *b)
```

```
{
```

```
int temp;
```

```
temp = *a;
```

```
*a=*b;
```

```
*b=temp;
```

```
printf("After swapping values in function a = %d, b = %d\n",*a,*b); // Formal parameters, a = 20, b = 10
```

```
}
```

Output

Before swapping the values in main a = 10, b = 20

After swapping values in function a = 20, b = 10

After swapping values in main a = 20, b = 10

Difference between call by value and call by reference

No.	Call by value	Call by reference
1	A copy of the value is passed into the function	An address of value is passed into the function
2	Changes made inside the function is limited to the function only. The values of the actual parameters do not change by changing the formal parameters.	Changes made inside the function validate outside of the function also. The values of the actual parameters do change by changing the formal parameters.
3	Actual and formal arguments are created at the different memory location	Actual and formal arguments are created at the same memory location

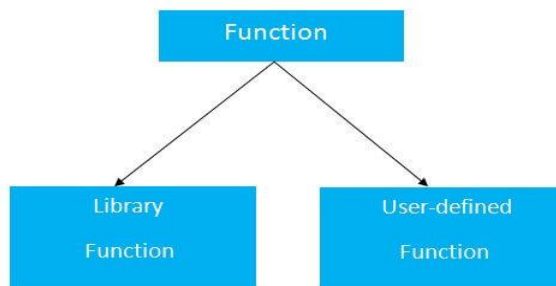
Advantage of functions

- By using functions, we can avoid rewriting same logic/code again and again in a program.
- We can call C 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.

Types of Functions

There are two types of functions in C programming:

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



Function Return Type

Function return type tells what type of value is returned after all function is executed. When we don't want to return a value, we can use the void data type.

Example: `int func(parameter_1,parameter_2);`

Types of Function According to Arguments and Return Value

Functions can be differentiated into 4 types according to the arguments passed and value returns these are:

1. Function with arguments and return value
2. Function with arguments and no return value
3. Function with no arguments and with return value
4. Function with no arguments and no return value

1.Function with arguments and return value

Syntax: Function declaration: `int function (int);`

Function call: `function(x);`

Function definition: `int function (int x)`

```
{  
    Statements;  
    return x;  
}
```

// C code for function with arguments

// and return value

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

```
#include <string.h>
```

```
int function(int, int[]);
```

```
int main()
```

```
{
```

```
    int i, a = 20;
```

```
    int arr[5] = { 10, 20, 30, 40, 50 };
```

```
    a = function(a, &arr[0]);
```

```
    printf("value of a is %d\n", a);
```

```
    for (i = 0; i < 5; i++)
```

```
    {
```

```
        printf("value of arr[%d] is %d\n", i, arr[i]);
```

```
    }
```

```
    return 0;
```

```
}
```

```
int function(int a, int* arr)
```

```
{
```

```
    int i;
```

```
    a = a + 20;
```

```
    arr[0] = arr[0] + 50;
```

```
    arr[1] = arr[1] + 50;
```

```
    arr[2] = arr[2] + 50;
```

```
    arr[3] = arr[3] + 50;
```

```
    arr[4] = arr[4] + 50;
```

```
    return a;
```

```
}
```

Output

value of a is 40

value of arr[0] is 60

value of arr[1] is 70

value of arr[2] is 80

value of arr[3] is 90

value of arr[4] is 100

2. Function with arguments but no return value

When a function has arguments, it receives any data from the calling function but it returns no values. These are void functions with no return values.

Syntax: Function declaration : `void function (int);`

Function call : `function(x);`

Function definition: `void function(int x)`

```
{
```

```
Statements;  
}
```

```
// C code for function  
// with argument but no return value  
#include <stdio.h>  
void function(int, int[], char[]);  
int main()  
{  
    int a = 20;  
    int ar[5] = { 10, 20, 30, 40, 50 };  
    char str[30] = "geeksforgeeks";  
    // function call  
    function(a, &ar[0], &str[0]);  
    return 0;  
}  
void function(int a, int* ar, char* str)  
{  
    int i;  
    printf("value of a is %d\n", a);  
    for (i = 0; i < 5; i++)  
    {  
        printf("value of ar[%d] is %d\n", i, ar[i]);  
    }  
    printf("\nvalue of str is %s\n", str);  
}
```

Output

```
value of a is 20  
value of ar[0] is 10  
value of ar[1] is 20  
value of ar[2] is 30  
value of ar[3] is 40  
value of ar[4] is 50  
value of str is geeksforgeeks
```

3. Function with no argument and no return value

When a function has no arguments, it does not receive any data from the calling function. Similarly, when it does not return a value, the calling function does not receive any data from the called function.

Syntax: Function declaration: void function();

Function call: function();

Function definition:

```
void function()  
{  
    statements;  
}
```

```
// C code for function with no  
// arguments and no return value  
#include <stdio.h>  
void value(void);  
void main() {  
    value();  
}  
void value(void)  
{  
    float year = 1, period = 5, amount = 5000, inrate = 0.12;  
    float sum;  
    sum = amount;  
    while (year <= period) {  
        sum = sum * (1 + inrate);  
        year = year + 1;  
    }  
    printf(" The total amount is :%f", sum);  
}
```

Output : The total amount is :8811.708984

4. Function with no arguments but returns a value

There could be occasions where we may need to design functions that may not take any arguments but returns a value to the calling function. An example of this is getchar function which has no parameters but it returns an integer and integer-type data that represents a character.

Syntax: Function declaration : int function();

Function call : function();

Function definition :

```
int function()
{
    statements;
    return x;
}
```

```
// C code for function with no arguments
// but have return value
#include <math.h>
#include <stdio.h>
int sum();
int main()
{
    int num;
    num = sum();
    printf("Sum of two given values = %d", num);
    return 0;
}
int sum()
{
    int a = 50, b = 80, sum;
    sum = sqrt(a) + sqrt(b);
    return sum;
}
```

Output

Sum of two given values = 16

Define Recursion

What is Recursion in C?

Recursion is the process of a function calling itself repeatedly till the given condition is satisfied. A function that calls itself directly or indirectly is called a recursive function and such kind of function calls are called recursive calls.

In C, recursion is used to solve complex problems by breaking them down into simpler sub-problems. We can solve large numbers of problems using recursion in C. For example, factorial of a number, generating Fibonacci series, generating subsets, etc.

Recursive Functions in C

In C, a [function](#) that calls itself is called Recursive Function. The recursive functions contain a call to themselves somewhere in the function body. Moreover, such functions can contain multiple recursive calls.

The basic syntax structure of the recursive functions is:

```
type function_name (args) {
    // function statements
    // base condition
    // recursion case (recursive call)
}
```

Example: C Program to Implement Recursion

```
// C Program to calculate the sum of first N natural numbers
// using recursion
#include <stdio.h>
int nSum(int n)
{
    // base condition to terminate the recursion when N = 0
    if (n == 0)
    {
        return 0;
    }

    // recursive case / recursive call
    int res = n + nSum(n - 1);
    return res;
}
```

```

}

int main()
{
    int n = 5;

    // calling the function
    int sum = nSum(n);

    printf("Sum of First %d Natural Numbers: %d", n, sum);
    return 0;
}

```

Output

Sum of First 5 Natural Numbers: 15

C program to print factorial of a given number using recursion

```

#include<stdio.h>
int factorial(int n)
{
    if(n==0)
        return 1;
    else
        return n*factorial(n-1);
}
int main ()
{
    int num;
    printf("Enter a number:");
    scanf("%d",&num);
    if(num<0)
        printf("factorial of negative number is not define");
    else
        printf("factorial of %d is %d",num,factorial(num));
    return 0;
}

```

Output :

Enter a number :5
Factorial of 5 is 120.

Pass Array to Functions With Program

In C, the whole array cannot be passed as an argument to a function. However, you can pass a pointer to an array without an index by specifying the array's name.

Arrays in C are always passed to the function as pointers pointing to the first element of the array. In C, we have three ways to pass an array as a parameter to the function. In the function definition, use the following

syntax: *return_type foo (array_type array_name[size], ...);*

Mentioning the size of the array is optional. So the syntax can be written as : *return_type foo (array_type array_name[], ...);*

In both of the above syntax, even though we are defining the argument as array it will still be passed as a pointer. So we can also write the **syntax as:** *return_type foo (array_type* array_name, ...);*

But passing an array to function results in [array decay](#) due to which the array loses information about its size. It means that the size of the array or the number of elements of the array cannot be determined anymore.

Example 1: Checking Size After Passing Array as Parameter

```

// C program to pass the array as a function and check its size
#include <stdio.h>
#include <stdlib.h>
// Note that arr[] for fun is just a pointer even if square
// brackets are used. It is same as
// void fun(int *arr) or void fun(int arr[size])
void func(int arr[8])
{
    printf("Size of arr[] in func(): %d bytes", sizeof(arr));
}

```



```
// Drive code
int main()
{
int arr[8] = { 1, 2, 3, 4, 5, 6, 7, 8 };
printf("Size of arr[] in main(): %d bytes\n", sizeof(arr));
func(arr);
return 0;
}
```

Output

Size of arr[] in main(): 32bytes
Size of arr[] in func(): 8 bytes

Structure as function arguments and structure as function values.

Structure as function arguments:

Structures can be passed as function arguments like all other data types. We can pass individual members of a structure, an entire structure, or a pointer to a structure to a function. Like all other data types, a structure or a structure member or a pointer to a structure can be returned by a function. Structure-function helps in writing better code.

//Structure as function arguments in C Programming

```
#include<stdio.h>

struct student
{
    char *name;
    int age;
    float per;
};

void display(struct student o)
{
    printf("\nName    : %s",o.name);
    printf("\nAge      : %d",o.age);
    printf("\nPercent   : %f",o.per);
}

int main()
{
    struct student o={"RAM",25,75.5};
    display(o);

    return 0;
}
```

Output

Name : RAM
Age : 25
Percent : 75.500000

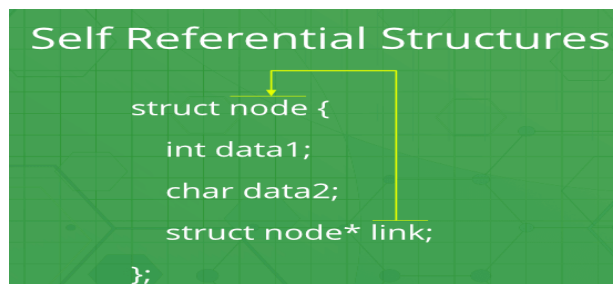
Structures containing Pointers

A structure pointer is defined as the [pointer](#) which points to the address of the memory block that stores a [structure](#) known as the structure pointer. Complex data structures like Linked lists, trees, graphs, etc. are created with the help of structure pointers. The structure pointer tells the address of a structure in memory by pointing the variable to the structure variable.

Self referential structures with examples.

The self-referential structure is a structure that points to the same type of structure. It contains one or more pointers that ultimately point to the same structure.

- Structures are a **user-defined** data structure type in C and C++.
- The main benefit of creating structure is that it can hold the different predefined data types.
- It is initialized with the struct keyword and the structure's name followed by this struct keyword.
- We can easily take the different data types like **integer value, float, char**, and others within the same structure.
- It reduces the complexity of the program.
- Using a single structure can hold the values and data set in a single place.
- In the C++ programming language, placing struct keywords is optional or not mandatory, but it is mandatory in the C programming language.
- Self-referential structure plays a very important role in creating other data structures like Linked list.
- In this type of structure, the object of the same structure points to the same data structure and refers to the data types of the same structure.
- It can have one or more pointers pointing to the same type of structure as their member.
- The self-referential structure is widely used in dynamic data structures such as **trees, linked lists**, etc.



Storage Classes –Auto, Register, Static ,Extern

Storage classes in C are used to determine the lifetime, visibility, memory location, and initial value of a variable. There are four types of storage classes in C

C language uses 4 storage classes

Storage classes in C				
Storage Specifier	Storage	Initial value	Scope	Life
auto	stack	Garbage	Within block	End of block
extern	Data segment	Zero	global Multiple files	Till end of program
static	Data segment	Zero	Within block	Till end of program
register	CPU Register	Garbage	Within block	End of block

Automatic

- Automatic variables are allocated memory automatically at runtime.
- The visibility of the automatic variables is limited to the block in which they are defined.
- The scope of the automatic variables is limited to the block in which they are defined.
- The automatic variables are initialized to garbage by default.
- The memory assigned to automatic variables gets freed upon exiting from the block.
- The keyword used for defining automatic variables is auto.

- Every local variable is automatic in C by default.

```
#include <stdio.h>
int main()
{
    int a; //auto
    char b;
    float c;
    printf("%d %c %f",a,b,c); // printing initial default value of automatic variables a, b, and c.
    return 0;
}
```

Output: garbage garbage garbage

Static

- The variables defined as static specifier can hold their value between the multiple function calls.
- Static local 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 global variable is limited to the file in which it has declared.
- The keyword used to define static variable is static.

```
#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); // the initial default value of c, i, and f will be printed.
}
```

Output: 0 0 0.000000 (null)

Register

- The variables defined as the register is allocated the memory into the CPU registers depending upon the size of the memory remaining in the CPU.
- We can not dereference the register variables, i.e., we can not use &operator for the register variable.
- The access time of the register variables is faster than the automatic variables.
- The initial default value of the register local variables is 0.
- The register keyword is used for the variable which should be stored in the CPU register. However, it is compiler's choice whether or not; the variables can be stored in the register.
- We can store pointers into the register, i.e., a register can store the address of a variable.
- Static variables can not be stored into the register since we can not use more than one storage specifier for the same variable.

```
#include <stdio.h>
int main()
{
    register int a; // variable a is allocated memory in the CPU register. The initial default value of a is 0.
    printf("%d",a);
}
```

Output:

0

External

- The external storage class is used to tell the compiler that the variable defined as extern is declared with an external linkage elsewhere in the program.
- The variables declared as extern are not allocated any memory. It is only declaration and intended to specify that the variable is declared elsewhere in the program.
- The default initial value of external integral type is 0 otherwise null.
- We can only initialize the extern variable globally, i.e., we can not initialize the external variable within any block or method.
- An external variable can be declared many times but can be initialized at only once.
- If a variable is declared as external then the compiler searches for that variable to be initialized somewhere in the program which may be extern or static. If it is not, then the compiler will show an error.

```
#include <stdio.h>
int main()
{
    extern int a;
    printf("%d",a);
}
Output
main.c:(.text+0x6): undefined reference to `a'
collect2: error: ld returned 1 exit status
```

Scope, visibility and lifetime of variables in functions

Scope: is defined as the availability of a variable inside a program, scope is basically the region of code in which a variable is *available* to use. There are four types of scope: **file scope, block scope, function scope and prototype scope.**

Visibility: of a variable is defined as if a variable is *accessible* or not inside a particular region of code or the whole program.

Lifetime of a variable: is the *time* for which the variable is taking up a *valid space* in the system's memory, it is of three types: **static lifetime, automatic lifetime and dynamic lifetime.**

Differentiate Local and External variable

Local Variables:

- Local variables in C are declared within a block of code, typically within a function or a compound statement.
- They are accessible only within the block of code in which they are declared.
- Local variables have local scope, meaning they cannot be accessed from outside the block of code in which they are declared.
- Local variables are created when the function is called and destroyed when the function exits.
- Each invocation of the function creates a new set of local variables, and changes made to these variables affect only the current invocation of the function.

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

```
void my_function() {
    int x = 10; // This is a local variable
    printf("%d\n", x);
}
int main() {
    my_function();
    // Attempting to access x here would result in an error since x is local to my_function
    return 0;
}
```

External Variables:

- External variables in C are declared outside of any function, typically at the beginning of a file or in a header file.
- They have global scope, meaning they can be accessed from anywhere within the program, including inside functions.
- External variables retain their value throughout the entire execution of the program.
- Modifications made to external variables are visible to all parts of the program that access them.

```
#include <stdio.h>
int x = 10; // This is an external (global) variable
void my_function()
```

```

{
    printf("%d\n", x); // Accessing the global variable x
}

int main()
{
    my_function(); // Output: 10
    return 0;
}

```

Global variable

In C programming, a **global variable** is a variable that is declared outside of any function and can be accessed by any function in the program. Unlike local variables, which are only accessible within their own function, global variables are visible to the entire program. To declare a global variable in C, we will simply declare outside of any function using the following syntax: **data_type variable_name;**

Passing the global variables as parameters using sample programs

```

#include <stdio.h>
// Global variable
int global_var = 10;
// Function that takes a global variable as a parameter
void print_global_var(int x) {
    printf("The value of global_var passed as parameter is: %d\n", x);
}

int main() {
    // Calling the function and passing the global variable as a parameter
    print_global_var(global_var);
    return 0;
}

```

Declaration and initialization of Pointers.

What is a Pointer in C?

A pointer is defined as a derived data type that can store the address of other C variables or a memory location. We can access and manipulate the data stored in that memory location using pointers.

Syntax of C Pointers :

The syntax of pointers is similar to the variable declaration in C, but we use the (*) **dereferencing operator** in the pointer declaration.

Syntax : **datatype * ptr;**

where

- **ptr** is the name of the pointer.
- **datatype** is the type of data it is pointing to.

1. Pointer Declaration

In pointer declaration, we only declare the pointer but do not initialize it. To declare a pointer, we use the (*) dereference operator **before its name**.

Example : **int *ptr;**

The pointer declared here will point to some random memory address as it is not initialized. Such pointers are called **wild pointers**.

2. Pointer Initialization

Pointer initialization is the process where we assign some initial value to the pointer variable. We generally use the (&) **addressof operator** to get the memory address of a variable and then store it in the pointer variable.

Example : **int var = 10;**

int * ptr;

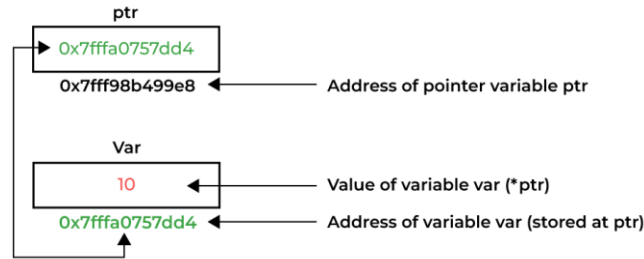
ptr = &var;

We can also declare and initialize the pointer in a single step. This method is called **pointer definition** as the pointer is declared and initialized at the same time.

Example : **int *ptr = &var;**

Pointer Dereferencing

Dereferencing a pointer is the process of accessing the value stored in the memory address specified in the pointer. We use the same (*) dereferencing operator that we used in the pointer declaration.



// C program to illustrate Pointers

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

```
void geeks()
```

```
{
```

```
    int var = 10;
```

```
    // declare pointer variable
```

```
    int* ptr;
```

```
    // note that data type of ptr and var must be same
```

```
    ptr = &var;
```

```
    // assign the address of a variable to a pointer
```

```
    printf("Value at ptr = %p \n", ptr);
```

```
    printf("Value at var = %d \n", var);
```

```
    printf("Value at *ptr = %d \n", *ptr);
```

```
}
```

```
// Driver program
```

```
int main()
```

```
{
```

```
    geeks();
```

```
    return 0;
```

```
}
```

Output

Value at ptr = 0x7fff1038675c

Value at var = 10

Value at *ptr = 10

Accessing the address of the a variable using & operator

Certainly! In C, you can use the **&** (address-of) operator to obtain the memory address of a variable. This operator returns the memory address where the variable is stored in the computer's memory.

Here's an example:

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

```
int main()
```

```
{
```

```
    int x = 10; // Printing the address of the variable x
```

```
    printf("Address of variable x: %p\n", (void*)&x);
```

```
    return 0;
```

```
}
```

Accessing the value of a variable through pointer

Certainly! In C, you can access the value of a variable through a pointer using the dereference operator *****. When you have a pointer pointing to a variable, dereferencing the pointer means accessing the value stored at the memory address that the pointer is pointing to.

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

```
int main()
```

```
{
```

```
    int x = 10;
```

```
    int *ptr = &x; // Pointer to integer, pointing to the address of variable x
```

```
    // Accessing the value of variable x through the pointer
```

```
    printf("Value of variable x (accessed through pointer): %d\n", *ptr);
```

```
    return 0;
```

```
}
```

Pointer Arithmetic

Pointer Arithmetic is the set of valid arithmetic operations that can be performed on pointers. The [pointer](#) variables store the memory address of another variable. It doesn't store any value.

Following arithmetic operations are possible on the pointer in C language:

1. Increment/Decrement of a Pointer
2. Addition of integer to a pointer
3. Subtraction of integer to a pointer
4. Subtracting two pointers of the same type
5. Comparison of pointers

1. Increment/Decrement of a Pointer

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.

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.

For Example:

```
#include <stdio.h>
// pointer increment and decrement
//pointers are incremented and decremented by the size of the data type they point to
int main()
{
    int a = 22;
    int *p = &a;
    printf("p = %u\n", p); // p = 6422288
    p++;
    printf("p++ = %u\n", p); //p++ = 6422292 +4 // 4 bytes
    p--;
    printf("p-- = %u\n", p); //p-- = 6422288      -4 // restored to original value

    float b = 22.22;
    float *q = &b;
    printf("q = %u\n", q); //q = 6422284
    q++;
    printf("q++ = %u\n", q); //q++ = 6422288      +4 // 4 bytes
    q--;
    printf("q-- = %u\n", q); //q-- = 6422284      -4 // restored to original value

    char c = 'a';
    char *r = &c;
    printf("r = %u\n", r); //r = 6422283
    r++;
    printf("r++ = %u\n", r); //r++ = 6422284      +1 // 1 byte
    r--;
    printf("r-- = %u\n", r); //r-- = 6422283 -1 // restored to original value

    return 0;
}
```

Output

```
p = 1441900792
p++ = 1441900796
p-- = 1441900792
q = 1441900796
q++ = 1441900800
q-- = 1441900796
r = 1441900791
r++ = 1441900792
r-- = 1441900791
```

2. Addition of Integer to Pointer

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.

```
// C program to illustrate pointer Addition
#include <stdio.h>
```

```
// Driver Code
int main()
{
    // Integer variable
    int N = 4;

    // Pointer to an integer
    int *ptr1, *ptr2;

    // Pointer stores the address of N
    ptr1 = &N;
    ptr2 = &N;

    printf("Pointer ptr2 before Addition: ");
    printf("%p \n", ptr2);

    // Addition of 3 to ptr2
    ptr2 = ptr2 + 3;
    printf("Pointer ptr2 after Addition: ");
    printf("%p \n", ptr2);

    return 0;
}
```

Output

```
Pointer ptr2 before Addition: 0x7ffca373da9c
```

```
Pointer ptr2 after Addition: 0x7ffca373daa8
```

3. 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.

```
// C program to illustrate pointer Subtraction
#include <stdio.h>
```

```
// Driver Code
int main()
{
    // Integer variable
    int N = 4;

    // Pointer to an integer
    int *ptr1, *ptr2;

    // Pointer stores the address of N
    ptr1 = &N;
    ptr2 = &N;

    printf("Pointer ptr2 before Subtraction: ");
    printf("%p \n", ptr2);

    // Subtraction of 3 to ptr2
    ptr2 = ptr2 - 3;
    printf("Pointer ptr2 after Subtraction: ");
    printf("%p \n", ptr2);

    return 0;
}
```

Output

```
Pointer ptr2 before Subtraction: 0x7ffd718ffebc
```

```
Pointer ptr2 after Subtraction: 0x7ffd718ffeb0
```


4. Subtraction of Two Pointers

The subtraction of two pointers is possible only when they have the same data type. The result is generated by calculating the difference between the addresses of the two pointers and calculating how many bits of data it is according to the pointer data type. The subtraction of two pointers gives the increments between the two pointers.

// C program to illustrate Subtraction

// of two pointers

#include <stdio.h>

// Driver Code

int 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);

return 0;

}

Output

ptr1 = 2715594428, ptr2 = 2715594424

Subtraction of ptr1 & ptr2 is 1

5. Comparison of Pointers

We can compare the two pointers by using the comparison operators in C. We can implement this by using all operators in C >, >=, <, <=, ==, !=. It returns true for the valid condition and returns false for the unsatisfied condition.

// C Program to illustrate pointer comparison

#include <stdio.h>

int main()

{

// declaring array

int arr[5];

// declaring pointer to array name

int* ptr1 = &arr;

// declaring pointer to first element

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.");

}

return 0;

}

Output

Pointer to Array Name and First Element are Equal

Precedence of address and de-referencing operators.

In C, the precedence of the address-of operator (&) and the dereference operator (*) is as follows:

1. **Dereference Operator (*):**
 - The dereference operator * has a higher precedence than almost all other operators in C.
 - It is used to obtain the value stored at the memory address pointed to by a pointer.
2. **Address-of Operator (&):**
 - The address-of operator & has a lower precedence than the dereference operator *.
 - It is used to obtain the memory address of a variable.

```
#include <stdio.h>
int main()
{
    int x = 10;
    int *ptr = &x; // Pointer to integer, pointing to the address of variable x

    // Example of using both operators:
    printf("Value of x: %d\n", *ptr); // Dereferencing the pointer to get the value of x
    printf("Address of x: %p\n", (void*)&x); // Getting the address of x

    return 0;
}
```

In this example, `*ptr` is dereferencing the pointer `ptr`, retrieving the value stored at the memory address it points to. `(void*)&x` is taking the address of variable `x` using the address-of operator `&`.

The precedence rules ensure that the dereference operation (*) takes place before the address-of operation (&).

Relationship between arrays and pointer

In C, an array and a pointer have a strong relationship. Here's how they are interconnected:

- An array's name represents the address of its first element. The array can be implicitly converted to a pointer pointing to the first element.
- Pointers can be used to access and modify elements in an array through pointer arithmetic.
- Both arrays and pointers enable indexed access to elements. However, the pointer uses arithmetic operations to calculate the memory address while the array uses a base address and an index.
- When a pointer is incremented, it moves to the next memory location based on the size of the data type it points to.

Accessing array elements using pointers

In C, you can use pointers to access array elements. When you declare an array in C, the name of the array behaves like a pointer to the first element of the array. Here's how you can access array elements using pointers:

```
#include <stdio.h>
int main()
{
    int arr[5] = {10, 20, 30, 40, 50};
    int *ptr = arr; // Pointer pointing to the first element of the array

    // Accessing array elements using pointer arithmetic
    printf("Array elements using pointers:\n");
    for (int i = 0; i < 5; i++) {
        printf("Element %d: %d\n", i, *(ptr + i)); // Dereferencing pointer
    }

    return 0;
}
```

Output

Array elements using pointers:

Element 0: 10
Element 1: 20
Element 2: 30
Element 3: 40
Element 4: 50

Pointers as function arguments

As an argument, a pointer is passed instead of a variable and its address is passed instead of its value. As a result, any change made by the function using the pointer is permanently stored at the address of the passed variable. In C, this is referred to as call by reference.

// C program to swap two values

// without passing pointer to

// swap function.

#include <stdio.h>

void swap(int* a, int* b)

{

int temp;

temp = *a;

***a = *b;**

***b = temp;**

}

// Driver code

int main()

{

int a = 10, b = 20;

printf("Values before swap function are: %d, %d\n",
a, b);

swap(&a, &b);

printf("Values after swap function are: %d, %d",
a, b);

return 0;

}

Output

Values before swap function are: 10, 20

Values after swap function are: 20, 10

Pointer arrays with example

In C, a pointer array is a homogeneous collection of indexed pointer variables that are references to a memory location. It is generally used in C Programming when we want to point at multiple memory locations of a similar data type in our C program. We can access the data by dereferencing the pointer pointing to it.

Syntax: pointer_type *array_name [array_size];

// C program to demonstrate the use of array of pointers

#include <stdio.h>

int main()

{

// declaring some temp variables

int var1 = 10;

int var2 = 20;

int var3 = 30;

// array of pointers to integers

int* ptr_arr[3] = { &var1, &var2, &var3 };

// traversing using loop

for (int i = 0; i < 3; i++) {

printf("Value of var%d: %d\tAddress: %p\n", i + 1, *ptr_arr[i], ptr_arr[i]);

}

return 0;

}

Output

Value of var1: 10 Address: 0x7fff1ac82484

Value of var2: 20 Address: 0x7fff1ac82488

Value of var3: 30 Address: 0x7fff1ac8248c

Differentiate between address and de-referencing operator

Address –of Operator(&)	Dereferencing Operator(*)
The address-of operator(&) is used to retrieve the memory address of variable	The Dereferencing operator(*) is used to access the values stored at a memory address pointed to by a pointer
It returns the memory address where the variable is stored in the computer's memory	It retrieves the value stored at a particular memory address
It is commonly used when you need to pass the address of a variable to a function or when working with pointers	It is commonly used when working with pointers to access or modify the data that the pointer is pointing to.
Int x=10 Int *ptr=&x;//Assigns the address of variable x to ptr	Int x=10; Int *ptr=&x;//ptr holds the address of variable x Int value=*ptr;// Retrieves the value stored at the address pointed to by ptr

Dynamic memory management functions with examples

The process of allocating memory at run time is known as **dynamic memory allocation**. Dynamic memory allocation in C language is possible by 4 functions of stdlib.h header file.

1. malloc()
2. calloc()
3. realloc()
4. free()

1. malloc() method

The “malloc” or “memory allocation” method in C is used to dynamically allocate a single large block of memory with the specified size, and returns a pointer to the first byte of the allocated space.

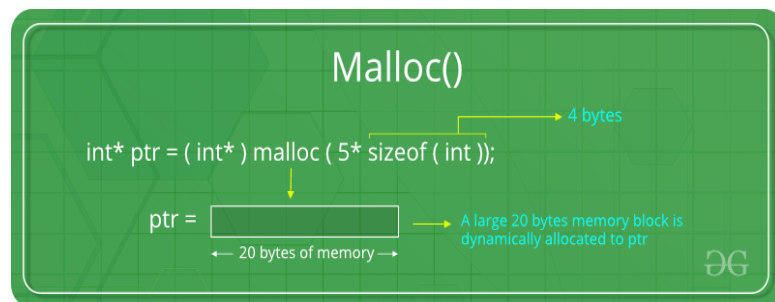
Ptr=(datatype)*malloc(specified size);

Ex: ptr=(int)*malloc(10);

It allocates 10 bytes of memory space to the pointer 'ptr' of type int and the base address is stored in the pointer variable ptr.

Ptr=(int)malloc(10*sizeof(int));

It allocates 10 times of space.



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

int main(){
    int n,i,*ptr,sum=0;
    printf("Enter number of elements: ");
    scanf("%d",&n);
    ptr=(int*)malloc(n*sizeof(int)); //memory allocated using malloc
    if(ptr==NULL)
    {
        printf("Sorry! unable to allocate memory");
        exit(0);
    }
    printf("Enter elements of array: ");
```

```

for(i=0;i<n;++i)
{
    scanf("%d",ptr+i);
    sum+=*(ptr+i);
}
printf("Sum=%d",sum);
free(ptr);
return 0;
}

```

Output

```

Enter elements of array: 3
Enter elements of array: 10
10
10
Sum=30

```

2. calloc() method

“calloc” or “contiguous allocation” method in C is used to dynamically allocate the specified number of blocks of memory of the specified type. it is very much similar to malloc() but has two different points and these are:

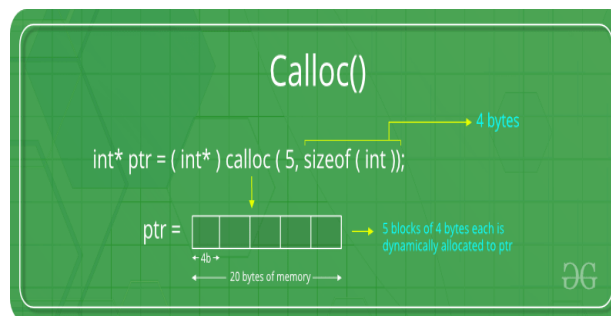
1. It initializes each block with a default value ‘0’.
2. It has two parameters or arguments as compare to malloc().
3. Allocates space for an array of elements, initializes them to zero and then return a pointer to the memory.

Syntax of calloc() in C

ptr = (cast-type*)calloc(n, element-size);

here, n is the no. of elements and element-size is the size of each element.

Ex: ptrt=(int*)calloc(5,2); // Allocates 5 blocks of memory,each block contains 2 bytes of memory



```

#include <stdio.h>
#include <stdlib.h>
int main()
{
    // This pointer will hold the
    // base address of the block created
    int* ptr;
    int n, i;
    // Get the number of elements for the array
    n = 5;
    printf("Enter number of elements: %d\n", n);
    // Dynamically allocate memory using calloc()
    ptr = (int*)calloc(n, sizeof(int));

    // Check if the memory has been successfully
    // allocated by calloc or not
    if (ptr == NULL) {
        printf("Memory not allocated.\n");
        exit(0);
    }
    else {
        // Memory has been successfully allocated
    }
}

```

```

printf("Memory successfully allocated using calloc.\n");

// Get the elements of the array
for (i = 0; i < n; ++i) {
    ptr[i] = i + 1;
}

// Print the elements of the array
printf("The elements of the array are: ");
for (i = 0; i < n; ++i) {
    printf("%d, ", ptr[i]);
}

return 0;
}

```

Output

Enter number of elements: 5

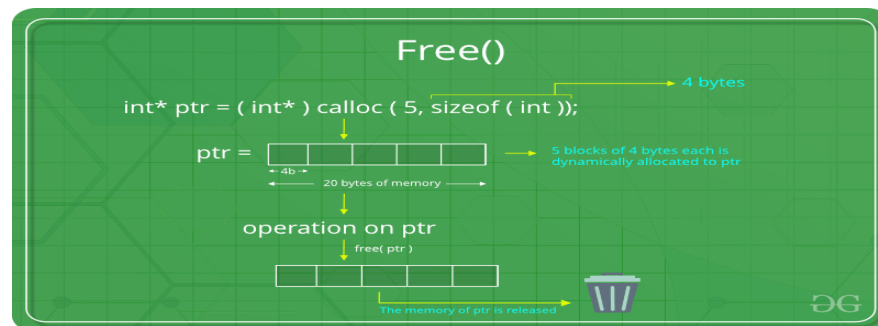
Memory successfully allocated using calloc.

The elements of the array are: 1, 2, 3, 4, 5,

3. free() method

“free” method in C is used to dynamically **de-allocate** the memory. The memory allocated using functions malloc() and calloc() is not de-allocated on their own. Hence the free() method is used, whenever the dynamic memory allocation takes place. It helps to reduce wastage of memory by freeing it.

Syntax of free() in C : free(ptr);



```

#include <stdio.h>
#include <stdlib.h>
int main()
{
    // This pointer will hold the
    // base address of the block created
    int *ptr, *ptr1;
    int n, i;

    // Get the number of elements for the array
    n = 5;
    printf("Enter number of elements: %d\n", n);

    // Dynamically allocate memory using malloc()
    ptr = (int*)malloc(n * sizeof(int));

    // Dynamically allocate memory using calloc()
    ptr1 = (int*)calloc(n, sizeof(int));

    // Check if the memory has been successfully
    // allocated by malloc or not
    if (ptr == NULL || ptr1 == NULL) {
        printf("Memory not allocated.\n");
        exit(0);
    }
    else {

```

```

// Memory has been successfully allocated
printf("Memory successfully allocated using malloc.\n");

// Free the memory
free(ptr);
printf("Malloc Memory successfully freed.\n");

// Memory has been successfully allocated
printf("\nMemory successfully allocated using calloc.\n");

// Free the memory
free(ptr1);
printf("Calloc Memory successfully freed.\n");
}

return 0;
}

```

Output

```

Enter number of elements: 5

Memory successfully allocated using malloc.

Malloc Memory successfully freed.

Memory successfully allocated using calloc.

Calloc Memory successfully freed.

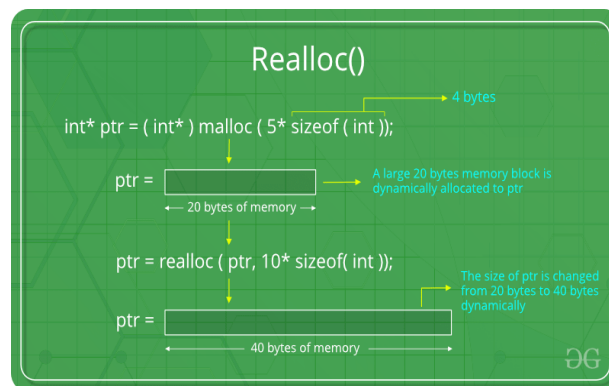
```

4. realloc() method

“realloc” or “re-allocation” method in C is used to dynamically change the memory allocation of a previously allocated memory. In other words, if the memory previously allocated with the help of malloc or calloc is insufficient, realloc can be used to **dynamically re-allocate memory**. re-allocation of memory maintains the already present value and new blocks will be initialized with the default garbage value.

Syntax of realloc() in C `ptr = realloc(ptr, newSize);`

where ptr is reallocated with new size 'newSize'.



```

#include <stdio.h>
#include <stdlib.h>
int main()
{
// This pointer will hold the
// base address of the block created
int* ptr;
int n, i;

// Get the number of elements for the array
n = 5;
printf("Enter number of elements: %d\n", n);

// Dynamically allocate memory using calloc()
ptr = (int*)calloc(n, sizeof(int));

// Check if the memory has been successfully

```

```

// allocated by malloc or not
if (ptr == NULL) {
    printf("Memory not allocated.\n");
    exit(0);
}
else {

    // Memory has been successfully allocated
    printf("Memory successfully allocated using calloc.\n");

    // Get the elements of the array
    for (i = 0; i < n; ++i) {
        ptr[i] = i + 1;
    }

    // Print the elements of the array
    printf("The elements of the array are: ");
    for (i = 0; i < n; ++i) {
        printf("%d, ", ptr[i]);
    }

    // Get the new size for the array
    n = 10;
    printf("\n\nEnter the new size of the array: %d\n", n);

    // Dynamically re-allocate memory using realloc()
    ptr = (int*)realloc(ptr, n * sizeof(int));

    // Memory has been successfully allocated
    printf("Memory successfully re-allocated using realloc.\n");

    // Get the new elements of the array
    for (i = 5; i < n; ++i) {
        ptr[i] = i + 1;
    }

    // Print the elements of the array
    printf("The elements of the array are: ");
    for (i = 0; i < n; ++i) {
        printf("%d, ", ptr[i]);
    }

    free(ptr);
}

return 0;
}

```

Output

```

Enter number of elements: 5

Memory successfully allocated using calloc.

The elements of the array are: 1, 2, 3, 4, 5,

Enter the new size of the array: 10

Memory successfully re-allocated using realloc.

The elements of the array are: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,

```


Structures containing pointers

A structure pointer is defined as the [pointer](#) which points to the address of the memory block that stores a [structure](#) known as the structure pointer. Complex data structures like Linked lists, trees, graphs, etc. are created with the help of structure pointers. The structure pointer tells the address of a structure in memory by pointing the variable to the structure variable.

Example:

```
// C program to demonstrate structure pointer
#include <stdio.h>
struct point
{
    int value;
};
int main()
{
    struct point s;
    // Initialization of the structure pointer
    struct point* ptr = &s;
    return 0;
}
```

Pointer to structure

In C, you can create pointers to structures just like you create pointers to other data types. Pointers to structures are useful when you want to dynamically allocate memory for structures or when you need to pass structures to functions efficiently. Here's an example demonstrating the use of pointers to structures:

```
#include <stdio.h>
#include <stdlib.h>
// Define a structure
struct Person {
    char name[50];
    int age;
};
int main() {
    // Declare a pointer to struct Person
    struct Person *personPtr;

    // Allocate memory for a struct Person object
    personPtr = (struct Person*)malloc(sizeof(struct Person));

    // Check if memory allocation was successful
    if (personPtr == NULL) {
        printf("Memory allocation failed\n");
        return 1; // Return error code
    }

    // Assign values to the fields of the structure using the pointer
    strcpy(personPtr->name, "John Doe"); // Assign name using string copy
    personPtr->age = 30;

    // Access and print the values using the pointer
    printf("Name: %s\n", personPtr->name);
    printf("Age: %d\n", personPtr->age);

    // Free the allocated memory
    free(personPtr);

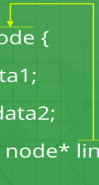
    return 0;
}
```

Self referential structure with examples

Self Referential structures are those [structures](#) that have one or more pointers which point to the same type of structure, as their member.

Self Referential Structures

```
struct node {  
    int data1;  
    char data2;  
    struct node* link;  
};
```



- Structures are a **user-defined** data structure type in C and C++.
- The main benefit of creating structure is that it can hold the different predefined data types.
- It is initialized with the struct keyword and the structure's name followed by this struct keyword.
- We can easily take the different data types like **integer value, float, char**, and others within the same structure.
- It reduces the complexity of the program.
- Using a single structure can hold the values and data set in a single place.
- In the C++ programming language, placing struct keywords is optional or not mandatory, but it is mandatory in the C programming language.
- Self-referential structure plays a very important role in creating other data structures like Linked list.
- In this type of structure, the object of the same structure points to the same data structure and refers to the data types of the same structure.
- It can have one or more pointers pointing to the same type of structure as their member.
- The self-referential structure is widely used in dynamic data structures such as **trees, linked lists**, etc.

Files and how to declare file pointer to a file.

File :

In C, a file is a collection of data stored on a storage medium such as a hard disk, SSD, or flash drive. In programming, a file is represented as a sequence of bytes. In order to work with files in C, you typically use the **FILE** data type and a set of file manipulation functions from the **stdio.h** library.

File Pointer :

A file pointer is a variable that is used to refer to an opened file in a C program. The file pointer is actually a structure that stores the file data such as the file name, its location, mode, and the current position in the file. It is used in almost all the file operations in C such as opening, closing, reading, writing, etc.

Syntax : FILE *ptr;

Declaring file pointer to a file

// C Program to demonstrate the file pointer

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

```
int main()
```

```
{
```

```
// declaring file pointer
```

```
FILE* fptr;
```

```
// trying to get the size of FILE datatype.
```

```
printf("Size of FILE Structure: %d bytes", sizeof(FILE));
```

```
return 0;
```

```
}
```

Output

Size of FILE Structure: 216 bytes

Types of Files in C

A file can be classified into two types based on the way the file stores the data. They are as follows:

- Text Files
- Binary Files

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.

C File Operations

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

1. Creating a new file – **fopen()** with attributes as “a” or “a+” or “w” or “w+”
2. Opening an existing file – **fopen()**
3. Reading from file – **fscanf()** or **fgets()**
4. Writing to a file – **fprintf()** or **fputs()**
5. Moving to a specific location in a file – [fseek\(\)](#), **rewind()**
6. Closing a file – **fclose()**

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:

Opening Modes	Description
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

Opening Modes	Description
	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.

As given above, if you want to perform operations on a binary file, then you have to append 'b' at the last. For example, instead of "w", you have to use "wb", instead of "a+" you have to use "a+b".

// C Program to illustrate file opening

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

```
#include <stdlib.h>
```

```
int main()
```

```
{
```

```
// file pointer variable to store the value returned by
```

```
// fopen
```

```
FILE* fptr;
```

```
// opening the file in read mode
```

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

```
// checking if the file is opened successfully
```

```
if (fptr == NULL)
```

```
{
```

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

```
exit(0);
```

```
}
```

```
return 0;
```

```
}
```

Output

The file is not opened. The program will now exit.

Concept of closing of file

Concept of file opening using various modes

Closing a File

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.

Syntax of fclose()

Ex : `fclose(file_pointer);`

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

Example:

```
FILE *fptr ;
fptr= fopen("fileName.txt", "w");
----- Some file Operations -----
fclose(fptr);
```

Example 1: Program to Create a File, Write in it, And Close the File

```
// C program to Open a File,
// Write in it, And Close the File
#include <stdio.h>
#include <string.h>
int main()
{
    // Declare the file pointer
    FILE* filePointer;
    // Get the data to be written in file
    char dataToBeWritten[50] = "GeeksforGeeks-A Computer Science Portal for Geeks";
    // Open the existing file GfgTest.c using fopen()
    // in write mode using "w" attribute
    filePointer = fopen("GfgTest.c", "w");
    // Check if this filePointer is null
    // which maybe if the file does not exist
    if (filePointer == NULL)
    {
        printf("GfgTest.c file failed to open.");
    }
    else
    {
        printf("The file is now opened.\n");
        // Write the dataToBeWritten into the file
        if (strlen(dataToBeWritten) > 0)
        {
            // writing in the file using fputs()
            fputs(dataToBeWritten, filePointer);
            fputs("\n", filePointer);
        }
        // Closing the file using fclose()
        fclose(filePointer);
        printf("Data successfully written in file GfgTest.c\n");
        printf("The file is now closed.");
    }
    return 0;
}
```

Output

```
The file is now opened.

Data successfully written in file GfgTest.c

The file is now closed.
```

Concept of Input/output operations on a file.

In C programming, input/output (I/O) operations on files are crucial for reading from and writing to files on a computer's file system. Here's an overview of how file I/O operations are typically performed in C:

Opening a File: To perform any I/O operation on a file, you first need to open it using the **fopen()** function. This function takes two arguments: the name of the file and the mode in which you want to open the file (e.g., read, write, append). For example:

```
FILE *file_ptr;
file_ptr = fopen("example.txt", "r"); // Opens the file for reading
if (file_ptr == NULL) {
    // Handle error if file couldn't be opened
}
```

Reading from a File: After opening the file for reading, you can use functions like **fscanf()** or **fgets()** to read data from the file. **fscanf()** is typically used to read formatted data, while **fgets()** is used for reading lines of text. For example:

```
char buffer[100];
fscanf(file_ptr, "%s", buffer); // Reads a string from the file
fgets(buffer, sizeof(buffer), file_ptr); // Reads a line of text from the file
```

Writing to a File: To write data to a file, you need to open it in write mode or append mode using `fopen()`, and then you can use functions like `fprintf()` or `fputs()` to write data to the file. For example:

```
fprintf(file_ptr, "%d\n", 42); // Writes an integer to the file
fputs("Hello, world!\n", file_ptr); // Writes a string to the file
```

Closing a File: After performing all necessary operations on the file, it's important to close it using the `fclose()` function. This ensures that any buffers associated with the file are flushed and that system resources are released. For example:

```
fclose(file_ptr);
```

Concept of random accessing files.

Random Access

A *random access file* in C is a kind of file that enables us to *write or read* any data in the disk file without having to read or write each section of data before it. In this file, we may instantly find for data, modify it, or even delete or remove it. We may open and close random access files in C in the same way we can sequential files, but we require a few more functions to do so.

Functions of Random Access file :Consequently, there are mainly three functions that assist in accessing the random access file in C:

- `ftell()`
- `rewind()`
- `fseek()`

The `ftell()` :function returns the current file position of the specified stream. We can use `ftell()` function to get the total size of a file after moving file pointer at the end of file. We can use `SEEK_END` constant to move the file pointer at the end of file.

Syntax: `long int ftell(FILE *stream)`

Example: File: `ftell.c`

`rewind()` function

The `rewind()` function sets the file pointer at the beginning of the stream. It is useful if you have to use stream many times.

Syntax: `void rewind(FILE *stream)`

Example: File: `file.txt`

`fseek()` function

The `fseek()` function is used to set the file pointer to the specified offset. It is used to write data into file at desired location.

Syntax: `int fseek(FILE *stream, long int offset, int whence)`

There are 3 constants used in the `fseek()` function for whence: `SEEK_SET`, `SEEK_CUR` and `SEEK_END`.

Different File handling Functions

File handling in C enables us to create, update, read, and delete the files stored on the local file system through our C program. The following operations can be performed on a file.

- Creation of the new file
- Opening an existing file
- Reading from the file

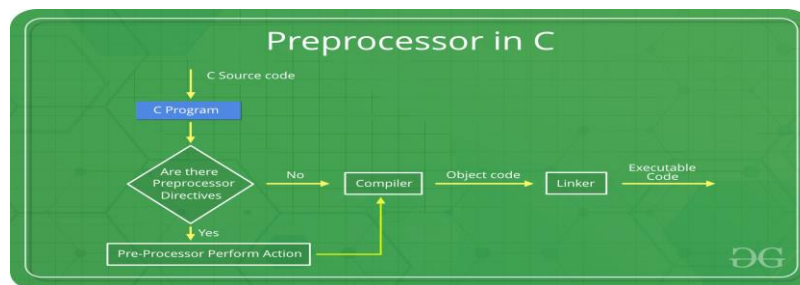
- Writing to the file
- Deleting the file

Functions for file handling

No.	Function	Description
1	fopen()	opens new or existing file
2	fprintf()	write data into the file
3	fscanf()	reads data from the file
4	fputc()	writes a character into the file
5	fgetc()	reads a character from file
6	fclose()	closes the file
7	fseek()	sets the file pointer to given position
8	fputw()	writes an integer to file
9	fgetw()	reads an integer from file
10	ftell()	returns current position
11	rewind()	sets the file pointer to the beginning of the file

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. We can place these preprocessor directives anywhere in our program.



Types of C Preprocessors

There are 4 Main Types of Preprocessor Directives:

1. Macros
2. File Inclusion
3. Conditional Compilation
4. 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`

// 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

```

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:

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>

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"

3. Conditional Compilation

[Conditional Compilation in C directives](#) is a type of directive that helps to compile a specific portion of the program or to skip the compilation of some specific part of the program based on some conditions. There are the following preprocessor directives that are used to insert conditional code:

1. **#if Directive**
2. **#ifdef Directive**
3. **#ifndef Directive**
4. **#else Directive**
5. **#elif Directive**
6. **#endif Directive**

#endif directive is used to close off the **#if**, **#ifdef**, and **#ifndef** opening directives which means the preprocessing of these directives is completed.

Syntax

```

#ifdef macro_name
    // Code to be executed if macro_name is defined
#ifndef macro_name
    // Code to be executed if macro_name is not defined
#if constant_expr
    // Code to be executed if constant_expression is true
#elif another_constant_expr
    // Code to be executed if another_constant_expression is true
#else
    // Code to be executed if none of the above conditions are true
#endif

```

//program to demonstrates the use of **#if**, **#elif**, **#else**,
// and **#endif** preprocessor directives.

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

```
// defining PI
```

```
#define PI 3.14159
```

```
int main()
```

```
{
```

```
#ifdef PI
```

```
    printf("PI is defined\n");
```

```
#elif defined(SQUARE)
```

```
    printf("Square is defined\n");
```

```
#else
```



```

    #error "Neither PI nor SQUARE is defined"
#endif

#ifndef SQUARE
    printf("Square is not defined");
#else
    cout << "Square is defined" << endl;
#endif

    return 0;
}

```

Output

PI is defined

Square is not defined

4. Other Directives

Apart from the above directives, there are two more directives that are not commonly used. These are:

1. **#undef Directive**
2. **#pragma Directive**

1. #undef Directive

The #undef directive is used to undefine an existing macro. This directive works as: **#undef LIMIT**

Using this statement will undefine the existing macro LIMIT. After this statement, every “#ifdef LIMIT” statement will evaluate as false

2. #pragma Directive

This directive is a special purpose directive and is used to turn on or off some features. These types of directives are compiler-specific, i.e., they vary from compiler to compiler.

Syntax : *#pragma directive*

Some of the #pragma directives are discussed below:

1. **#pragma startup:** These directives help us to specify the functions that are needed to run before program startup (before the control passes to main()).
2. **#pragma exit:** These directives help us to specify the functions that are needed to run just before the program exit (just before the control returns from main()).

Need of Preprocessor directives

Preprocessor directives are essential in C for several reasons:

1. **Header Files Inclusion:** Preprocessor directives are commonly used to include header files (**#include**) that contain declarations of functions, constants, and macros used in a C program. This allows for modularity and reusability of code, as commonly used functions and definitions can be stored in separate header files and included wherever needed.
2. **Conditional Compilation:** Preprocessor directives such as **#ifdef**, **#ifndef**, **#if**, **#else**, and **#endif** enable conditional compilation of code based on compile-time conditions. This allows developers to include or exclude certain parts of the code based on platform-specific requirements, debugging needs, or feature toggles.
3. **Macro Definitions:** Preprocessor directives like **#define** are used to define macros, which are symbolic names representing a sequence of code. Macros are extensively used for creating constants, defining inline functions, and simplifying complex expressions. They offer flexibility and enhance code readability and maintainability.
4. **File Inclusion Guard:** Preprocessor directives like **#ifndef**, **#define**, and **#endif** are often used together to create file inclusion guards in header files. This prevents multiple inclusions of the same header file in a single translation unit, which helps avoid issues such as redefinition errors and excessive compilation time.
5. **Debugging and Logging:** Preprocessor directives can be utilized to conditionally enable or disable debugging statements (**#ifdef DEBUG**) or logging messages based on compile-time flags. This allows developers to include debug information in development builds while excluding it from release builds, improving performance and reducing code size.
6. **Platform-Specific Code:** Preprocessor directives are frequently employed to write platform-specific code that behaves differently on different operating systems or architectures. By using conditional compilation directives, developers can maintain a single codebase while adapting it to various platforms.